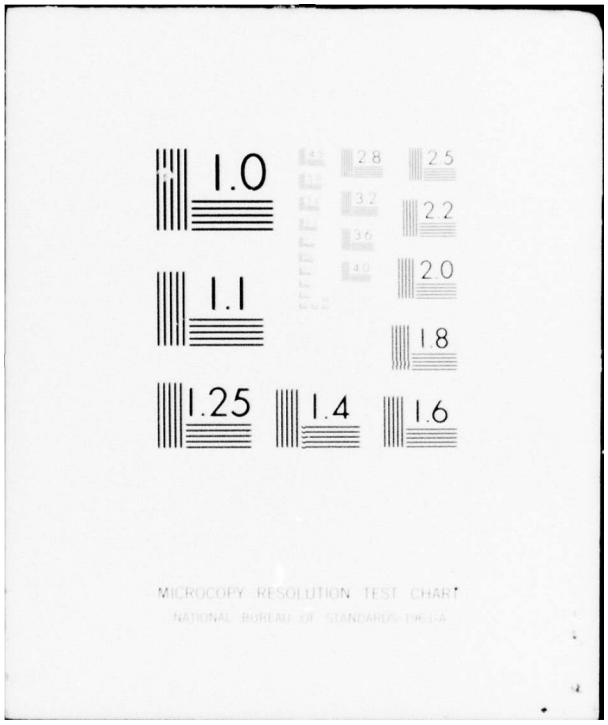
AD-A042 646 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
MEMORY ORGANIZATIONS AND THEIR EFFECTIVENESS FOR MULTIPROCESSIN--ETC(U)
MAY 77 F A BRIGGS DAAB07-72-C-0259
UNCLASSIFIED R-768 NL

1 OF 3
AD
A042646


1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

REPORT R-768 MAY, 1977 UILU-ENG 77-2215

COORDINATED SCIENCE LABORATORY

MEMORY ORGANIZATIONS AND THEIR EFFECTIVENESS FOR MULTIPROCESSING COMPUTERS

FAYÉ ALAYÉ BRIGGS

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): MEMORY ORGANIZATIONS AND THEIR EFFECTIVENESS FOR MULTIPROCESSING COMPUTERS,	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER: R-768, UILU-ENG-77-2215
7. AUTHOR(s): Fayé Alayé/Briggs	8. CONTRACT OR GRANT NUMBER(s): DAAB 07-72-C-0259, NSF-MCS-73-03488/A01
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 240p.
11. CONTROLLING OFFICE NAME AND ADDRESS: Joint Services Electronics Program	12. REPORT DATE: May 1977
	13. NUMBER OF PAGES: 198
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Memory Organization
Interleaved Memories
Multiporcessor Memory
Memory Access Conflict

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Organizations of interleaved multimodule semiconductor memories are studied to facilitate accessing of memory words by parallel-pipelined multiple instruction stream processors. All memory modules are assumed to be identical and are characterized by the address cycle (address hold time) and memory cycle of a and c time units respectively. A total of N (= 2^n) memory modules are arranged such that there are ℓ (= 2^b) lines for addresses and m(= 2^{n-b}) memory modules per line.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 097 700

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

For a parallel-pipelined processor of order (s, p), which consists of p parallel processors each of which is a pipelined processor with s degrees of multiprogramming, there can be up to s·p memory requests in each instruction cycle. The memory interference problems which arise in such systems are investigated.

Performance is evaluated as a function of the memory configuration (ℓ, m), the module characteristics (a,c), and the processor order (s, p). Results show that for reasonably large values of N, high performance can be obtained even in the nonbuffered case when ℓ is a·p or more. Buffering has its maximum effect on performance when ℓ is near a·p. When ℓ must be greater than a·p for adequate performance in the nonbuffered case, buffering can be used to reduce ℓ while maintaining performance.

Some design tradeoffs were discussed and examples were given to illustrate the wide variety of design options that can be obtained.

UILU-ENG 77-2215

MEMORY ORGANIZATIONS AND THEIR EFFECTIVENESS FOR MULTIPROCESSING COMPUTERS

by

Fayé Alayé Briggs

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259 and in part by the National Science Foundation under Grant MCS 73-03488 A01.

Approved for public release. Distribution unlimited.

MEMORY ORGANIZATIONS AND THEIR EFFECTIVENESS FOR MULTIPROCESSING COMPUTERS

FAYÉ ALAYÉ BRIGGS
B. Eng., Ahmadu Bello University, 1971
M.S., Stanford University, 1974

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor E. S. Davidson

Urbana, Illinois

MEMORY ORGANIZATIONS AND THEIR EFFECTIVENESS FOR MULTIPROCESSING COMPUTERS

Fayé Alayé Briggs, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1977

Organizations of interleaved multimodule semiconductor memories are studied to facilitate accessing of memory words by parallel-pipelined multiple instruction stream processors. All memory modules are assumed to be identical and are characterized by the address cycle (address hold time) and memory cycle of a and c time units respectively. A total of $N (= 2^n)$ memory modules are arranged such that there are $\ell (= 2^b)$ lines for addresses and $m (= 2^{n-b})$ memory modules per line.

For a parallel-pipelined processor of order (s, p), which consists of p parallel processors each of which is a pipelined processor with s degrees of multiprogramming, there can be up to $s \cdot p$ memory requests in each instruction cycle. The memory interference problems which arise in such systems are investigated.

Performance is evaluated as a function of the memory configuration (ℓ, m), the module characteristics (a, c), and the processor order (s, p). Results show that for reasonably large values of N, high performance can be obtained even in the nonbuffered case when ℓ is $a \cdot p$ or more. Buffering has its maximum effect on performance when ℓ is near $a \cdot p$. When ℓ must

be greater than a·p for adequate performance in the nonbuffered case, buffering can be used to reduce ℓ while maintaining performance.

Some design tradeoffs were discussed and examples were given to illustrate the wide variety of design options that can be obtained.

ACKNOLWEDGEMENTS

The author wishes to express his gratitude to his advisor, Dr. Edward Davidson, for his guidance, encouragement and helpful suggestions throughout this thesis. Dr. Davidson's many hours spent in reading and refining this thesis are appreciated.

The author is also grateful to his collegues at the Coordinated Science Laboratory for creating a vivifying intellectual atmosphere.

TABLE OF CONTENTS

		Page
1.	BACKGROUND AND MOTIVATION	1
1.1.	Introduction	1
1.2.	Processor Organization	4
1.3.	Timing Characteristics of Memory Module	11
1.4.	Some Previous Models	22
1.5.	Problem Statement	25
1.6.	Overview of Dissertation	25
2.	THE L-M MEMORY ORGANIZATION	27
2.1.	Introduction	27
2.2.	Memory Configuration	28
2.3.	Memory Request Scheduling	35
2.4.	Processor-Memory Interconnection	37
3.	PERFORMANCE ANALYSIS OF L-M MEMORY ORGANIZATION	40
3.1.	Introduction	40
3.2.	State Diagrams for p = 1	42
3.3.	State Reduction and Line Decomposition	70
3.4.	Line State Space	86
3.5.	Probability of Acceptance, $P_A(a, c, p)$	92
3.6.	Bounds on $P_A(a, c, p)$	111
4.	SIMULATION OF BUFFERED AND NONBUFFERED REQUESTS	117
4.1.	Introduction	117
4.2.	Nonbuffered Request Processor System	123
4.3.	Buffered Request Processor System	125
4.4.	Discussion	128
5.	ANALYSIS OF RESULTS	133
5.1.	Introduction	133
5.2.	Effect of Number of Modules (N) on Performance	134
5.3.	Effect of the Number of Lines on Performance	139
5.4.	Effect of Module Characteristics on Performance	142
5.5.	Effect of Processor Order on Performance	147
5.6.	Effect of Processor Speed on Performance	150
5.7.	Effect of Buffering on Performance	156
5.8.	Design Tradeoffs	165
5.9.	Burst Mode Operation	180

6. CONCLUSIONS..... 187

6.1 Summary of Results..... 187
6.2 Suggestions for Further Research..... 189

APPENDIX A..... 191

LIST OF REFERENCES..... 196

VITA..... 198

1. BACKGROUND AND MOTIVATION

1.1 Introduction

In the quest for higher performance in computer systems, two architectural techniques, namely, parallelism and pipelining, evolved to enhance the computation capability of the systems. In addition to the architectural alternatives, higher performance may also be achieved by increasing the switching speed of the electronic components. These three methods of improving the performance are not necessarily mutually exclusive. Although performance may be improved by the above techniques, it may be degraded considerably if the memory system is organized inefficiently and does not match the processor system in speed. Furthermore, a very efficient memory organization for multiprocessor systems may be cost prohibitive. These factors have prompted extensive investigation into techniques for organizing memories for multiprocessor systems.

In some highly parallel processor systems, concurrency is achieved by the multiplicity of independent processing units which execute separate instruction streams on separate data streams [1]. However, there exist other highly parallel computer systems, like ILLIAC IV, which are characteristically array processors that perform the same computation on a large collection of related data elements simultaneously [2]. In this research, parallel processors will refer to the former organization.

Parallelism or concurrency of instructions and data transfers also occurs in pipelined computers which have become common of recent.

Pipelining is a technique of decomposing a sequential process into a sequence of computation steps, each of which can be processed in a special functionally dedicated and autonomous unit, called a segment which operates concurrently with other segments. Hence a pipelined processor is composed of segments which are arranged so that consecutive steps of an instruction can be assigned to distinct segments of the pipeline for processing.

One form of pipelining occurs in the highly partitioned and overlapped instruction execution technique implemented in the IBM 360/91, which achieves a high efficiency by the concurrency of instructions and data transfer [3]. Another form of pipelining is the "stream" pipelining which performs the same arithmetic operation on a series of operands as they flow through the pipe. Examples of these are the CDC STAR-100 [4], and TI-ASC [5].

Such highly concurrent processors will be characterized and a description of the general model of the processor organization will be presented in the next section.

In a multiprocessor environment, main memory is a prime system resource which is usually shared by all the processors. Hence care must be taken in the organization of the memory system to avoid severe performance degradation due to memory interference caused by two or more processors simultaneously attempting to access the same module of the memory system.

It would be undesirable to have one monolithic unit of memory to be shared among several processors, as this would result in serious memory interference, hence the memory is partitioned into several

independent memory modules. This scheme resolves interference by allowing simultaneous access to more than one module but considerable interference can still result if the memory addresses are contiguous within a module. Interleaving of memory modules is used to alleviate this problem.

In most highly concurrent computer systems, interleaving of memory modules is also required in order to obtain a balance between effective processor and memory cycles. For example, the CDC STAR-100 has a processor cycle of 40ns and a memory cycle of 1280ns [4]. At most, one memory reference can be made in one processor cycle. For such fast processors, the rate at which data can be transferred between processors and main memory is often limited by the transfer capabilities of the memory itself and the memory busses. Hence the memory is usually organized to meet the memory bandwidth requirements of the system. The memory bandwidth is the rate at which memory can transfer information, usually represented in words per second.

On the other hand, some processors, such as microprocessors, exhibit processor cycles which are usually slower than the memory cycles of some memories. In such cases, the memories are usually underutilized, unless the processors are organized to create a balance between the processor and memory cycles.

In the past, magnetic memories have been used in the main memory systems of multiprocessor systems. However, with the advent of large scale integrated circuits, semiconductor memories have been playing increasingly important roles in the synthesis of main memory systems. Their inherent modularity makes them very appealing in the design of

multimodule memory systems for multiprocessor systems. In addition to their flexibility and nondestructive readout capability, the cost per bit of semiconductor memories remain virtually constant as the module size increases or decreases for a wide range of module sizes [6]. In contrast, the cost per bit of magnetic memories, such as ferrite core, rises very rapidly for decreasing module size. Furthermore, some current semiconductor memories exhibit timing characteristics which may be exploited to enhance the data transfer capabilities of multiple instruction stream computer systems. These timing characteristics will be discussed in section 1.3.

Hence in this research, we describe a method for exploiting the capabilities of semiconductor memories to obtain an effective multimodule memory organization for parallel-pipelined multiple instruction stream processors. Furthermore, the memory interference problem in such processor-memory systems are investigated.

1.2 Processor Organization

The concept of multiprocessors has been introduced in section 1.1. The processor organization to be discussed in this section is a theoretical model chosen to include a broad class of multiple instruction stream multiple data stream (MIMD) processors [1]. A formal definition of the pipelined processor model adopted here is given below.

Definition 1.2.1 A pipelined processor of order s is modeled as an ordered set of s segments $(s_0, s_1, \ldots, s_{s-1})$, each of which can simultaneously be processing a distinct step or phase of a distinct instruction. □

Once an instruction is initiated in a segment, it flows from segment to segment for its execution, where each segment performs a specific suboperation on a distinct phase of the instruction. It is considered that each segment has an output latch or register to help retain its autonomy. Figure 1.2.1 shows a nonpipelined processor as one monolithic unit, and Figure 1.2.2 illustrates a pipelined processor of order 3.

The pipelined processor defined above can be implemented in two different ways, namely, as a single instruction stream and multiple instruction stream pipelined processors. The following definition will be of aid in understanding the two different implementations.

Definition 1.2.2 The rth process or instruction stream, I(r) is a sequence of instructions that require execution. Thus,

$$I(r) = \alpha_{1r}, \alpha_{2r}, \alpha_{3r}, \ldots.$$

where α_{ij} = ith instruction from the jth instruction stream. □

Figure 1.2.3 is a space-time illustration of one form of pipelining processor of order 6. In this scheme, execution of instructions from the same stream are overlapped. The problems usually associated with the single instruction stream pipelined processors are the performance degradation and control problems due to data dependencies and branch instructions.

In this research, multiple instruction stream pipelined processor organizations were adopted, since performance degradation and control problems due to data dependencies and branch instructions are absent.

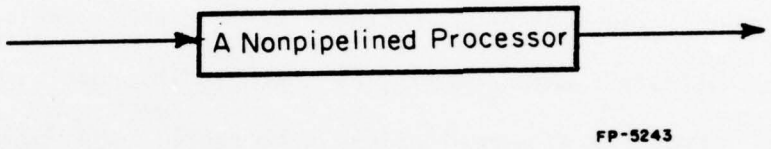


Figure 1.2.1 A nonpipelined processor as a monolithic unit

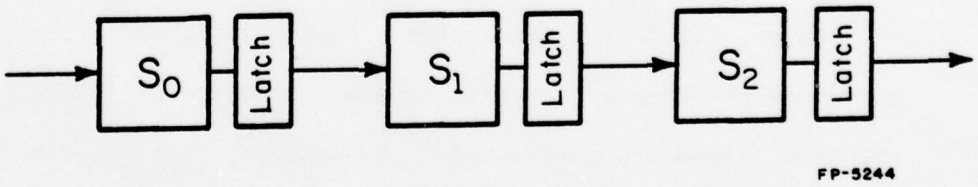


Figure 1.2.2 A pipelined processor of order 3

Concurrency in such processors is achieved only between distinct instruction streams as illustrated in Figure 1.2.4. In this scheme each instruction is partitioned into s distinct phases and each distinct phase is sequentially assigned to each distinct segment. In general, s distinct processes are in execution concurrently and if an instruction from a process is initiated at time instant t, the next instruction from the same process will be initiated at time instant t + s. Hence there is no execution overlap between instructions from the same stream.

Notice in Figure 1.2.4 that all instructions have identical flow patterns. Pipelines in which all instructions have identical flow patterns are termed single function pipelines. On the other hand, in a multifunction pipeline, there are two or more distinct flow patterns and each instruction may use one of these flow patterns [7]. In this research it is assumed that the pipeline processor is a single function pipeline.

In a pipelined processor of order s, s separate instructions will be in different phases of their execution steps. These s instructions are assumed to come from distinct instruction streams as in [8]. Thus the degree of multiprogramming, for a pipelined processor of order s, is also s.

The pipelined processor can be partitioned so that each segment takes the same time to complete its execution step.

Definition 1.2.3 One segment time unit (STU), is the time, in seconds required by each segment to execute each distinct phase of an instruction. □

Segment

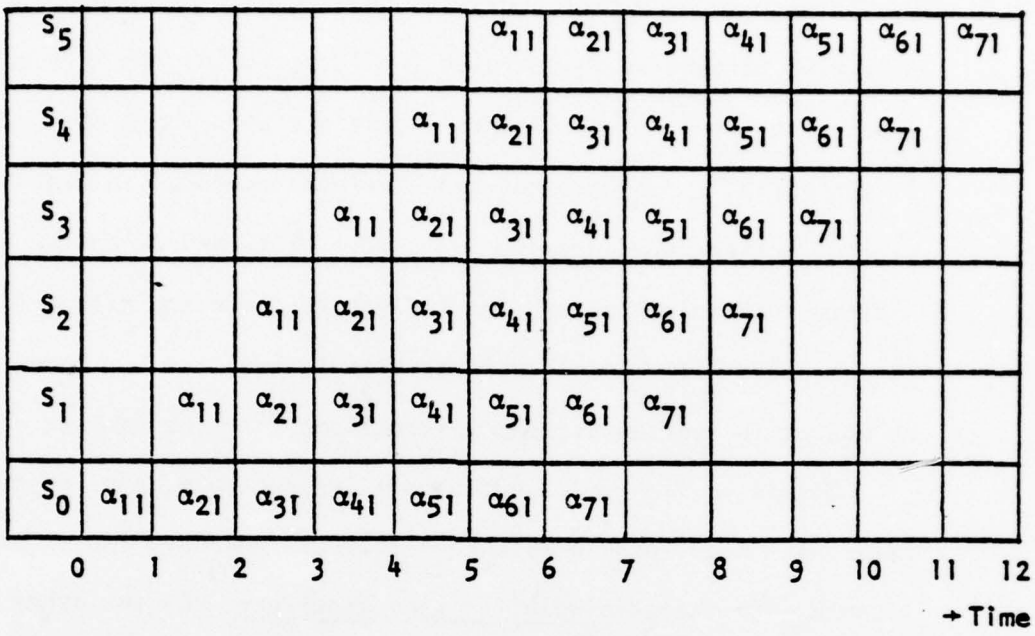


Figure 1.2.3 Single instruction stream processing in a pipelined processor of order 6.

Segment

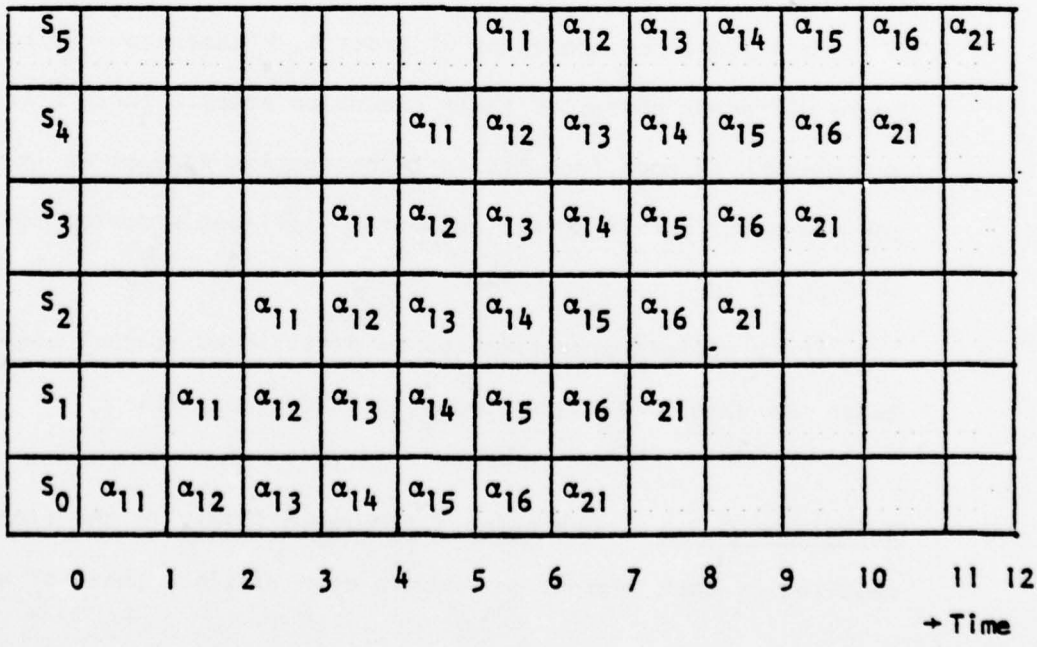


Figure 1.2.4 Multiple instruction stream processing in a pipelined processor of order 6.

Hence if the phases of an instruction are partitioned so that it takes τ seconds to execute each phase of the instruction, then one segment time unit is equal to τ seconds.

Assume also that a pipelined processor of order s can issue one memory request per STU, hence, s memory requests can be issued in one instruction cycle; where one instruction cycle = $s \cdot \tau$ seconds. Notice that the instruction cycle is fixed irrespective of the instruction. Hence a pipelined processor is characterized by s and τ, the degree of pipelining and the segment time unit respectively. From now on, all time units will be expressed as an integer number of STUs, unless otherwise stated.

A reservation table [9], used to illustrate the flow of computation through the segments of a pipeline, is shown in Figure 1.2.5 for a straight-through pipelined processor of order 6.

Following initiation of an instruction process at time instant t, an x in cell (u, v) indicates that a task requires the segment associated with row u for segment time interval $<t + v, t + v + 1>$. Note that the reservation table for the examples of single and multiple instruction stream pipelined processors, illustrated in Figure 1.2.3 and 1.2.4 respectively, would be identical to that shown in Figure 1.2.5.

The generalized processor organization will now be discussed.

Definition 1.2.4 A parallel pipelined processor of order (s,p) [10] is modeled as a set of p identical and independent, but synchronized processors, each of which is a pipelined processor of order s. □

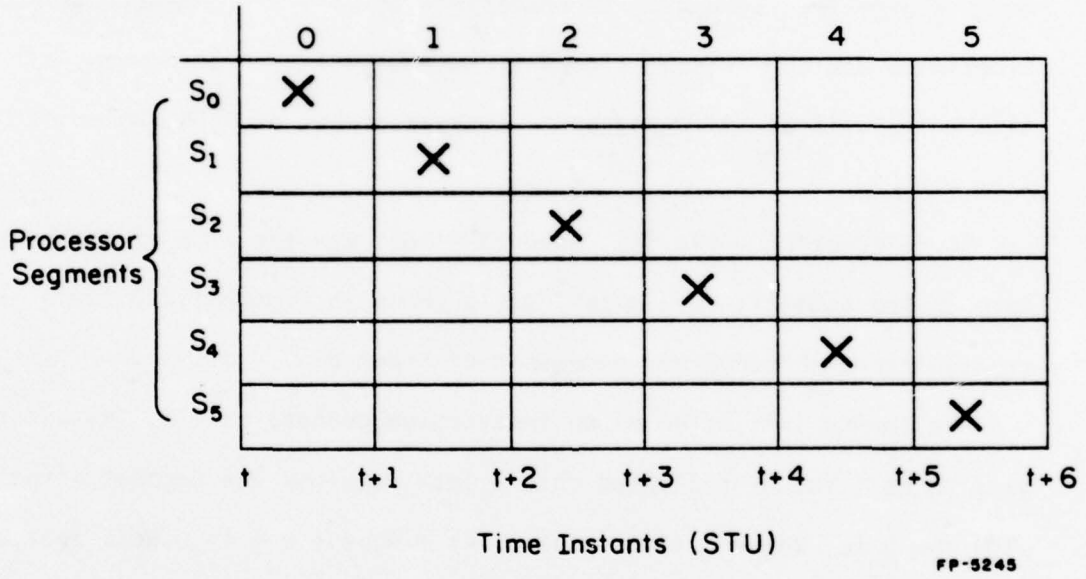


Figure 1.2.5 Reservation table of a straight-through pipelined processor of order 6

Figure 1.2.6 illustrates various configurations of parallel-pipelined processors.

A parallel-pipelined processor is thus completely specified by the degree of pipelining, s, the parallelism, p, and the segment time, τ. A parallel-pipelined processor of order (s, p) can issue p simultaneous memory requests each STU. Hence, it can execute $s \cdot p$ distinct instruction streams concurrently.

The processor bandwidth, B_p, defined as the number of memory requests that can be issued per second, will be adopted as a performance measurement of the parallel-pipelined processor of order (s, p). Hence, $B_p = \frac{p}{\tau}$. Note that s is not explicitly involved in the definition of B_p.

1.3 Timing Characteristics of Memory Module

In this section, the timing characteristics of the memory module are discussed to highlight the features which are exploited in the organization of the memory modules for parallel-pipelined processors.

In section 1.1, the inherent modularity and cost-effectiveness of semiconductor memories were pointed out. The cost-effectiveness of semiconductor memories is even much better over core memories for small module sizes. These factors account for the increasing use of LSI memories, instead of core memories for implementing a flexible multimodule memory organization. Hence the LSI memory chip is discussed to highlight the block structures.

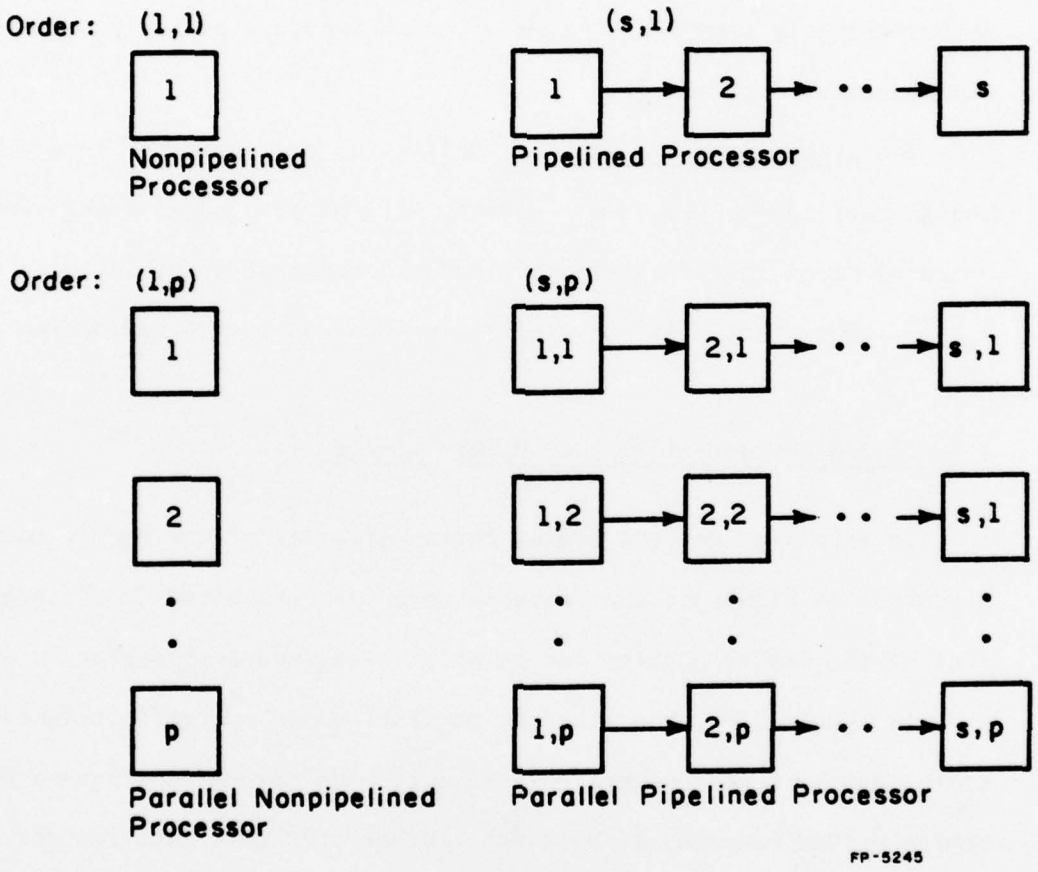


Figure 1.2.6 Configurations of parallel-pipelined processors

The LSI memory chips are assumed to be fully decoded, fixed address random access memory [6]. Typically, read only memory chips consist of four functional blocks, as shown in Figure 1.3.1. They include an address register to latch in the address, an address decoder to select the addressed word, the ROM matrix and an output data buffer stage. In some memory chips, the address register and the output data buffer are not fabricated on the memory chip. However, with recent developments in LSI technology, the cost of adding these buffers on the memory chip at fabrication is insignificant. The corresponding simplified timing diagram for a ROM chip is shown in Figure 1.3.2 and time units are represented in seconds. Assume for simplicity that the chip select and address signals are gated into the chip simultaneously. Although in practice, the chip select signal may precede or follow the address signal depending on the timing characteristics of the chip. The output data is usually valid and available after the memory access time, t_{ac}, of the chip. The duration of the output data, t_{do}, depends on whether the storage elements of the chip are static or dynamic. In practice, t_{do} is controlled as to keep nonvalid output off the data bus.

In the read and write RAM chip, there are typically five principal functional blocks as shown in Figure 1.3.3. These include the address register and decoder, the storage cells and the input and output data buffers. A typical timing diagram for the read cycle of the RAM chip is identical to the ROM's timing diagram in Figure 1.3.2. However, Figure 1.3.4 shows the timing diagram for the write cycle of the RAM chip.

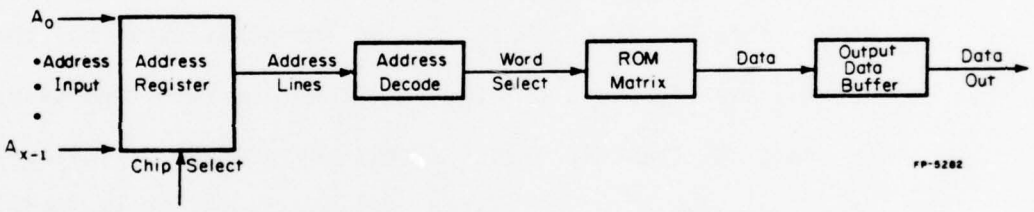


Figure 1.3.1 Functional blocks of Read Only memory chip

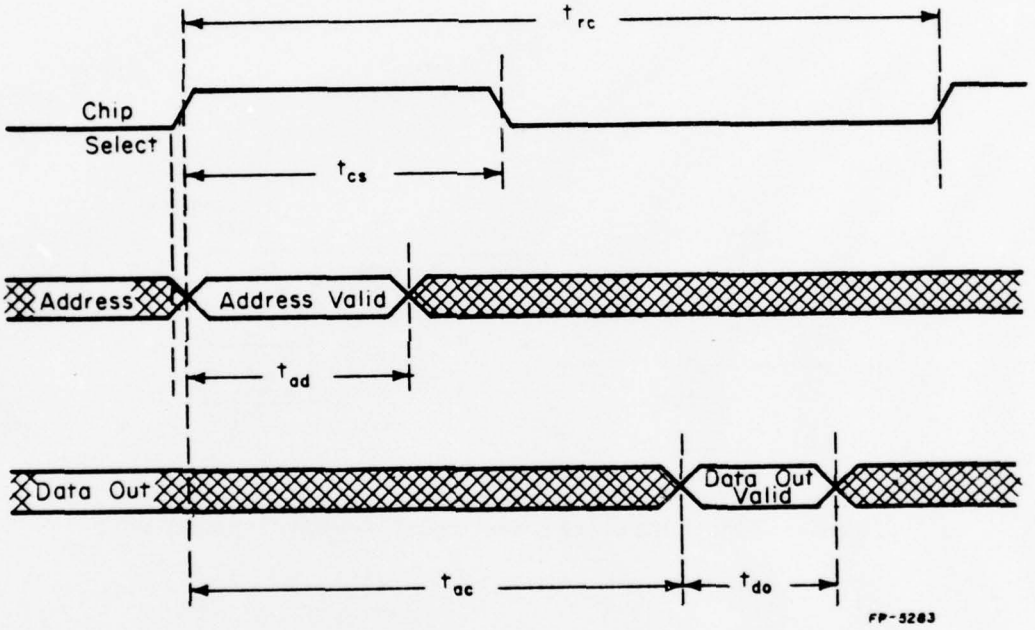


Figure 1.3.2 Timing diagram for ROM chip

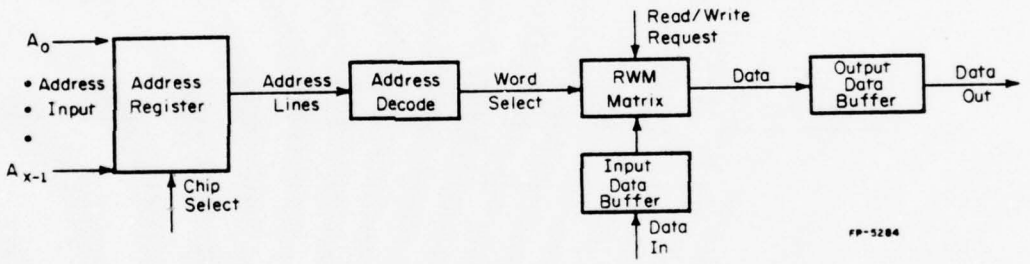


Figure 1.3.3 Functional blocks of read/write RAM chip

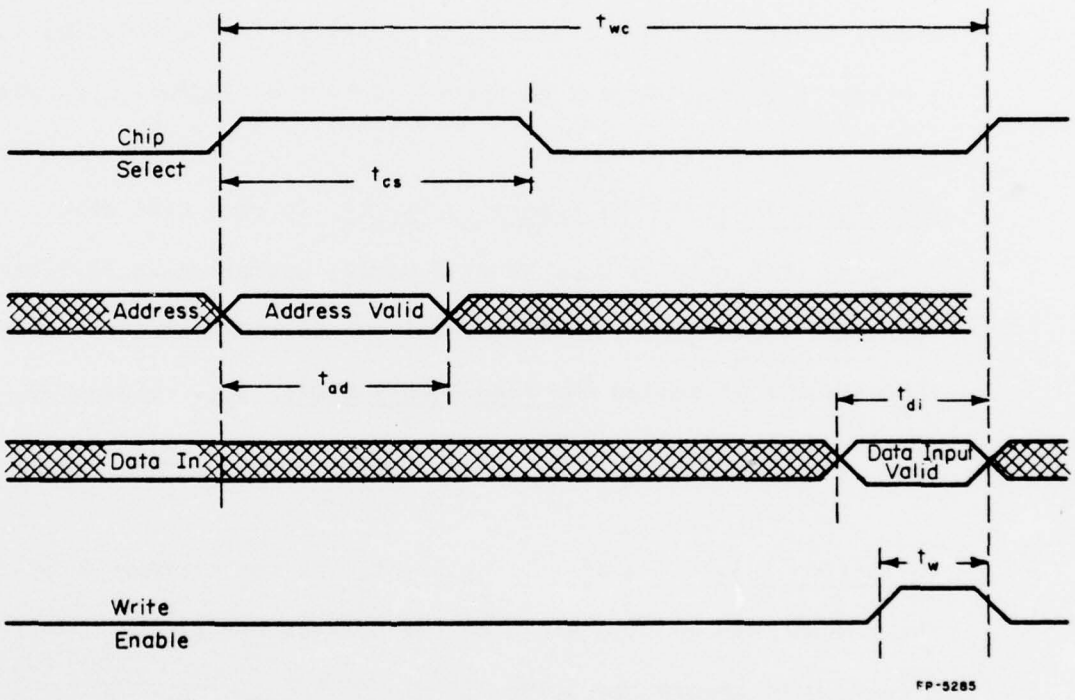


Figure 1.3.4 Timing diagram for write cycle of RAM chip

Let t_{cs} and t_{ad} be the chip select pulse width and the address pulse width respectively. Furthermore, t_{di} and t_w denote the data input pulse width and the write enable pulse width respectively. In simple terms, a memory operation is initiated by broadcasting an address to the address bus for t_{ad} seconds, to the address register of the memory chip, whereupon the address is further decoded in the decoder block, to select the corresponding memory bit. The selected memory bit contents are then altered as specified by the read/write request function.

Definition 1.3.1 The memory cycle, t_c, is the time that a memory chip remains busy after a memory operation is initiated. For read memory operation, this the minimum duration between successive read requests and is called the read memory cycle, t_{rc}. Similarly, for write operation, this is called the write memory cycle, t_{wc}. □

Definition 1.3.2 The address cycle, t_a, is the minimum duration that the address can be maintained on the address bus of the memory chip for a successful memory operation. □

Hence in Figures 1.3.2 and 1.3.4, the address cycle, $t_a = t_{ad}$. The end of the input data pulse need not coincide with the end of the write memory cycle shown in Figure 1.3.4. Usually, the input data can be presented to the data bus and gated by the write enable pulse after the memory access time, t_{ac}. The chip select pulse width and the address cycle for the write operation are usually identical to those of the read operation. However, for some memory chips they need not be so.

The timing constraints are generally obvious from the simplified timing diagram for the ROM and RAM chips. However, in general, $t_{rc} \leq t_{wc}$ for RAM chips. Furthermore, for any memory chip, $t_a \leq t_c$. Since the read memory cycle, t_{rc} may not always equal the write memory cycle, t_{wc}, we can for analytical purposes, introduce an effective memory cycle which takes into consideration the distribution of read and write memory requests. For simplicity assume that the proportion of read and write requests are f_r and f_w respectively. Then,

$$f_r + f_w = 1 .$$

Hence the effective memory cycle of the RAM chip is

$$t_{ec} = f_r\, t_{rc} + f_w\, t_{wc} .$$

Similarly, if the address cycles for read and write memory operations differ, the effective address cycle can be obtained.

Definition 1.3.3 The memory bandwidth of a RAM chip is

$$B_c = \frac{1}{t_{ec}} ,$$

which is the request servicing capacity of the chip. □

Typically, each semiconductor memory chip realizes some $z = 2^x$ words of 1 bit each. A memory module of z words of some w bits each is then usually organized by interconnecting w chips in a one dimensional array, as in Figure 1.3.5, so that, amongst other things, corresponding chip select leads, write enable signals, and addresses are common.

In effect, a memory module has three separate and independent busses, namely, the address bus, the data-input bus, and the data-output bus.

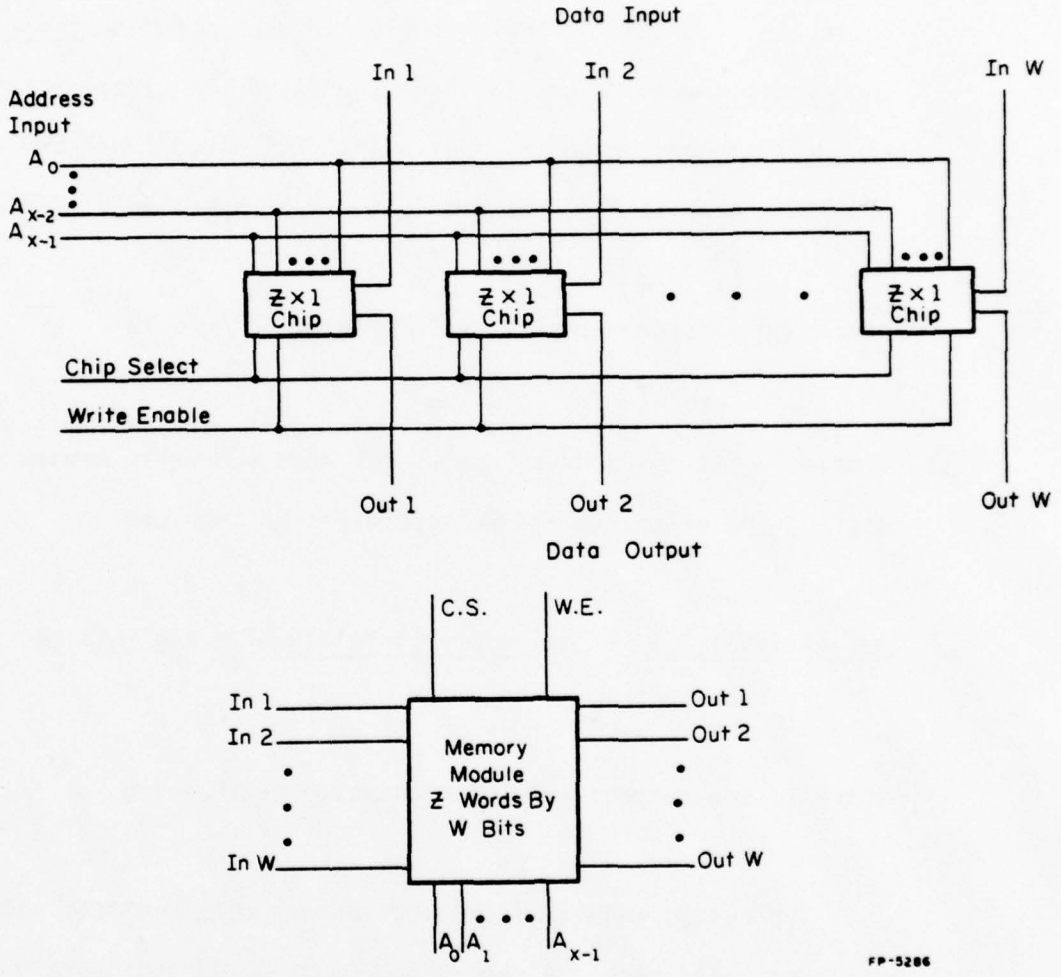


Figure 1.3.5 Memory module organization

Since a memory organization for highly parallel processors is investigated here, only maximally parallel bus structure [11] will be considered for the memory system. In such bus structures, simultaneous operations can be in progress on the three separate and independent busses.

Since all the chips, which are assumed to be identical, act simultaneously in a memory module, the memory bandwidth of a module is identical to the memory bandwidth of a chip in the module.

Definition 1.3.4 The memory bandwidth, B_m, of a fixed address RAM module is

$$B_m = B_c = \frac{1}{t_{ec}},$$

which indicates the request servicing capacity of the module. □

For RAM memory chips and hence modules, the address is held on the address bus at least as long as data is held on the data bus. That is,

$$0 < t_{do} \leq t_{ad} \quad \text{and} \quad 0 < t_{di} \leq t_{ad}.$$

Hence the data busses do not pose a limiting constraint on how often addresses can be dispatched to the address bus and are therefore not considered here explicitly.

In summary, a memory module is characterized by its address and memory cycles, (t_a, t_c). They are referred to as the absolute memory module characteristics, because the cycles, t_a and t_c, are expressed in seconds. However, the address and memory cycles can be quantized such that they are both expressed as integer numbers of STUs, namely,

$a = \left\lceil \frac{t_a}{\tau} \right\rceil$ and $c = \left\lceil \frac{t_c}{\tau} \right\rceil$, where τ is the segment time unit in seconds. Hence the relative module characteristics are (a, c). For example, if τ = 50ns, t_a = 140ns and t_c = 240ns, the relative module characteristics are (3, 5). Since in general, $t_a \leq t_c$, then $a \leq c$. Henceforth, the relative module characteristics will be referred to simply as the module characteristics unless otherwise stated.

1.4 Some Previous Models

Various analytic and simulation models have been developed to evaluate the performance of a multiprocessor computer system. In multiprocessor systems, several tasks are executed concurrently. These tasks may generate requests to memory simultaneously. Multiple requests sometimes result in memory conflicts despite the interleaving of the memory modules.

All models presented here assume some form of memory interleaving and evaluate the access conflict problem.

Hellerman [12] presented a model in which a single stream of intermixed independent instructions and data requests was scanned in the order of their arrival until the first duplicate memory module number was found. These first K distinct requests are then granted in parallel. For N interleaved memory modules, the average memory bandwidth for Hellerman's model is

$$B = \sum_{K=1}^{N} \frac{K^2(N-1)!}{N^K(N-K)!} .$$

He found that a good numerical approximation of the above expression

when $1 \leq N \leq 45$ is $N^{0.56} \approx \sqrt{N}$, accurate to within 4%.

Knuth and Rao [13] reduced Hellerman's expression into a closed form,

$$B = \sqrt{\frac{\pi N}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2N}} + O(N^{-1}) .$$

Burnett and Coffman [14, 15] extended Hellerman's model by separating the instruction requests from the data requests and showed that the system bandwidth can be increased considerably because of the locality in programs due to the sequential nature of instructions. This was modeled by introducing two parameters, namely, α and β, where α is the probability of a request addressing the next module in sequence (modulo N) and $\beta = (1-\alpha)/(N-1)$, is the probability of addressing each other module out of sequence. It was found analytically that the bandwidth increases exponentially with increase in α.

The above models seem to assume a single processor with instruction look-ahead capabilities, hence they will be called overlap-processor models. One of the first few analytic models for a multiprocessor system was proposed by Skinner and Asher [16]. The analysis made use of a discrete Markov chain model and was limited to a small number of processors (≤ 2) because of the complexity involved for larger systems.

Strecker [17] investigated the conflict problem in a multiprocessor system with p processors and N memory modules in which the processor cycle, that is, the segment time unit, is equal to the pulse width of the output data in a read memory cycle. By approximate analysis, a closed form representation of the bandwidth was obtained as

$$N[1-(1-\frac{1}{N})^p] .$$

Ravi [18] has a similar model which he analyzed using combinatorials to arrive at the bandwidth as

$$\sum_{K=1}^{t} \frac{K \cdot K!\ S(p,K) \binom{N}{K}}{N^p},$$

where $t = \min(p, N)$ and $S(p, K)$ are Stirling numbers of the second kind. It is interesting to note that Strecker's formula is a closed form representation of Ravi's formula. Bhandarkar [19] expanded on Strecker's results. Sastry and Kain [20] had similar models but also investigated the performance using different storage allocations for instructions and data with interleaving only in the instruction space. Baskett and Smith [21] have also investigated the memory conflict problem in multiprocessor systems.

One common characteristic of the multiprocessor models discussed above with respect to the parallel-pipelined processor proposed here is that the multiprocessor systems are all parallel-pipelined processors of order (1, p) having access to an N module memory system with module characteristics (a, c) = (1, 1). In this research, the memory conflict problem and hence the bandwidth is investigated for more general multiprocessor models, namely, systems encompassing a wide variety of parallel-pipelined processors of order (s, p), (ℓ, m) interleaved memory configurations and memory module characteristics (a, c). The parameters s, p, ℓ, m, a, c are chosen such that $1 \leq a \leq c$, $1 \leq c \leq s$, $p \geq 1$, and ℓ and m are nonnegative integer powers of 2 and $\ell m = 2^n = N$.

1.5 Problem Statement

The purpose of this research is:

(i) To investigate the effect of memory interference on system performance for a variety of relative module characteristics (a, c) and memory configurations (ℓ, m).

(ii) To study the effect of buffering memory requests when conflict occurs.

(iii) To give some methodology and design tradeoffs for a cost effective memory configuration.

(iv) To obtain guidelines for desirable semiconductor memory organization for parallel-pipelined multiple instruction stream processors.

1.6 Overview of Dissertation

So far, the background materials that motivated this research has been presented. In Chapter 2, the memory organization is developed. The module access conditions are also outlined. In Chapter 3, the performance of the memory organization is investigated for the parallel-pipelined processor of order (s, p). Discrete Markov models are developed to aid in the performance analysis of the case where p = 1. State reduction and line decomposition techniques are introduced to reduce the complexity and size of the state diagrams developed for p = 1. In this chapter, it was also shown that the complexity of the performance analysis grows with $\left\lfloor \frac{c-1}{a} \right\rfloor$ for a > 1. Using the line state transition

diagram for $p = 1$, closed form solutions of the probability of acceptance for $p > 1$ are obtained for a class of module characteristics. Furthermore, bounds on the performance are also obtained for any set of module characteristics. In Chapter 4, simulation models are developed for two different processor schemes, namely, non-buffered and buffered request processor systems. In particular, these schemes highlight two possible ways of handling requests when conflict occurs. In Chapter 5, the effects of the various parameters on performance are investigated. In addition, some design trade-offs are studied. The burst mode of operation, which dispatches simultaneously all requests issued in one memory cycle at the end of the cycle, is compared to the multiplex mode of operation, which dispatches requests as they are issued, that is, every segment time unit. Chapter 6 presents overall conclusions and prospects for further research.

2. THE L-M MEMORY ORGANIZATION

2.1 Introduction

In the previous chapter, a semiconductor random access memory chip was characterized by its address and memory cycle times, t_a and t_c respectively. Many large scale integrated RAM chips have their address cycle significantly smaller than the memory cycle time. This difference is increased when an address latch is incorporated within the chip to gate-in the address. Similarly, write data hold time may be short and read data may be enabled in a controllable window. It was assumed that the address is typically held on the address bus at least as long as data is held on the data bus. Hence the data busses do not pose a limiting constraint and are not explicitly considered in the memory organization discussed here. In this discussion, a line is used to denote an address bus within the memory. Hence, assuming that there are N identical memory modules in the memory, there can be up to N independent lines in the memory.

In this chapter, a memory organization is developed to exploit the capabilities of the memory chips discussed in Chapter 1. For a system environment with s·p distinct instruction streams running concurrently, the absolute size of the main memory M, would be expected to be large enough to accomodate at least the working set [22] of the s·p distinct processes. Since the research focuses on the memory conflict problem, page faulting is not modeled here.

With respect to present day practice, the total number of words z, in a memory module is small. Hence for a large main memory of size M, the number of memory modules N will probably be large, since M = Nz. The memory organization for multiprocessor systems discussed by previous authors have N lines for N independent memory modules. Although the performance of such memory organization is good, the line cost is usually high. Assuming that each module contains an address latch, a module in which a memory operation is initiated uses its associated line for a duration much less than one memory cycle per access. Therefore, more than one module can share a line thereby increasing the line utilization and reducing the line cost. However, as a consequence of line-sharing, the performance is degraded. Furthermore, a more complex bussing scheme is required as additional conflict situations are introduced. In the one-dimensional memory organization (that is, one module per line), the memory conflict problem arises when two or more simultaneous memory requests reference the same memory module hence the same line. For the two-dimensional memory discussed here, a conflict may also occur when a memory request references a busy line or a busy module on a line. In general, the degree of performance degradation for an associated degree of line-sharing is not intuitively obvious. Hence, we will study the tradeoffs and relationships of line-sharing and performnace.

2.2 Memory Configuration

The memory and processor operation described so far suggest that

the memory system can be partitioned in such a way as to accommodate the possible arrival rate of p requests per STU for a parallel-pipelined processor of order (s, p). Each stream of the s·p streams in the processor issues one request every s STUs. The memory system can be developed from the basic topology of memory system configuration for a multiprocessor system in which the number of lines and the number of memory modules are equal. It is obvious that because of the multiplicity of processing elements, more than one line is usually necessary to avoid excessive memory conflicts. If the segment time and the address and memory cycle times are all equal, then no line sharing is required because whenever a module is active, its associated line is also active. However, if $\tau < t_a < t_c$, then whenever a module is active for a memory cycle t_c, its associated line is active for only a fraction of the memory cycle. By multiplexing a group of modules on a line, the period for which the line is inactive may be used to broadcast the address of a new request which references an inactive module on the line. Hence some lines can be eliminated and their associated modules equally distributed and muitiplexed with modules on other lines. In the memory organization to be discussed, the modules are organized in a two-dimensional matrix, with line i and module j on line i referred to as L_i and $M_{i,j}$ respectively. This organization, referred to as the L-M memory organization, is shown in Figure 2.2.1. The memory organization consists of $N(= 2^n)$ identical memory modules arranged such that there are ℓ lines and m modules per line, where $\ell = 2^b$ for integer b and n such that $0 \leq b \leq n$ and $m = 2^{n-b}$. Recall that a line refers to the

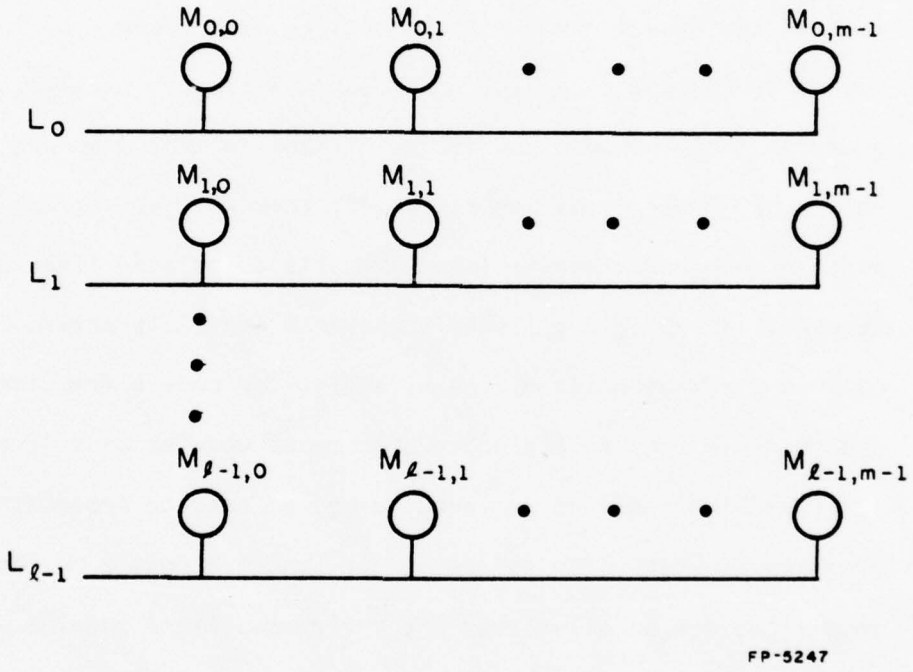


Figure 2.2.1 L-M memory organization

address bus common to a set of m modules. Figure 2.2.2 shows the bus structures of the memory organization. Each set of modules on a line in addition to sharing the same address bus share the same data input and data output busses. That is, there is one each of address, data input and data output busses for the set of m memory modules on each line.

Given the memory address of a word in memory, as shown in Figure 2.2.3, the least significant n bits of the address determine the line and module segments that are required to access the word, and the higher order bits of the address determine the addresss of the word within the selected module. The b least significant bits of the n bits address one of the 2^b lines, L_i, and the next higher order n - b bits address one of the 2^{n-b} modules on line i, $M_{i,j}$. Hence, the modules are interleaved on the low order n bits and the lines on the low order b bits. This scheme, as will be shown in the next chapter, tries to maximize the probability that for p = 1, a successive requests are to distinct lines and c successive requests are to distinct modules.

Figure 2.2.4 shows the reservation table for a parallel-pipelined processor of order (s, p) = (7, 1) having access to line L_i and module $M_{i,j}$ of a memory system whose module characteristics are (a, c) = (2, 4). Any computation initiated at time t runs straight through the 7 processor segments during $<t, t + 7>$ and uses some line segment throughout $<t + 1, t + 3>$ and some module segment on that line throughout $<t + 1, t + 5>$. Processor segments S_1, S_2, S_3, and S_4 may be used simply to retain the states of the processes as they access memory. Information

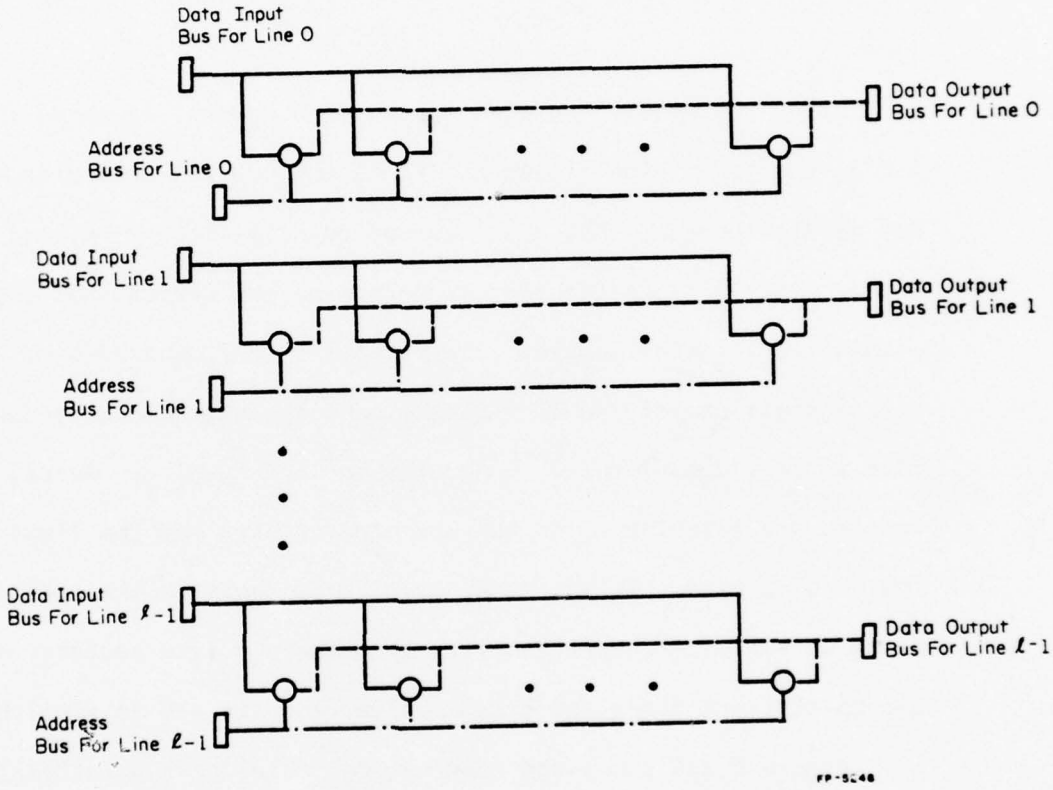


Figure 2.2.2 Bus structures of the memory organization

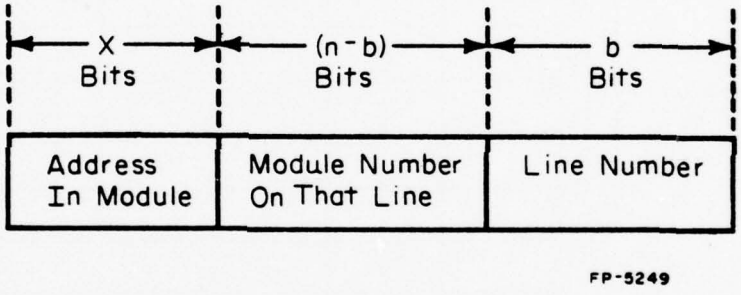


Figure 2.2.3 Memory address word

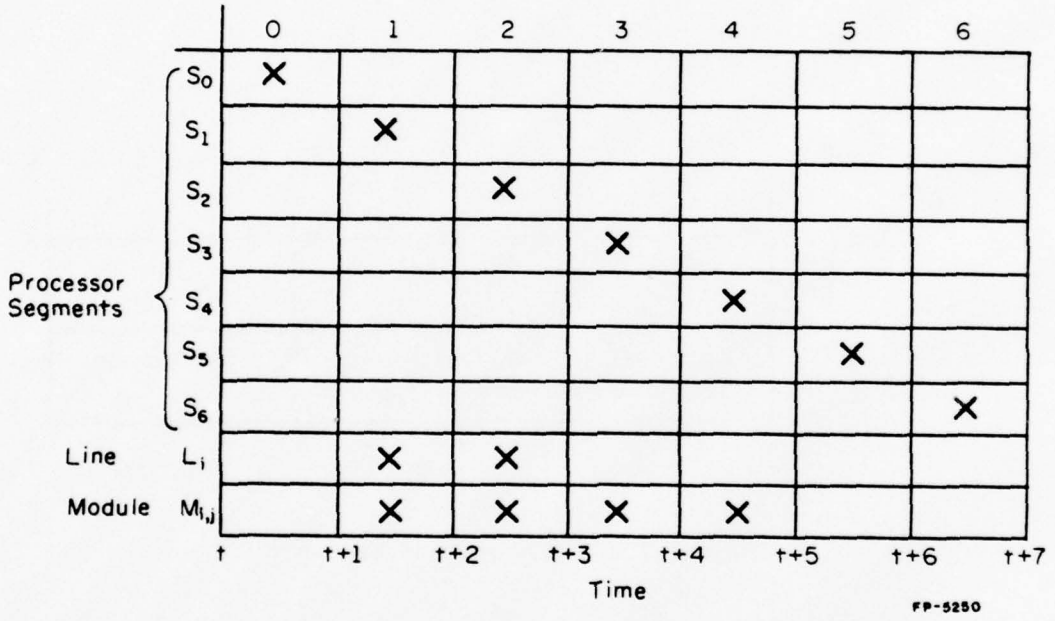


Figure 2.2.4 Reservation table for parallel-pipelined processor of order (7, 1)

from these segments can also be used to control certain functions in the memory system. Note that the operations in the line and module segments cannot be preempted. All tasks executed by the processor are identical in this respect except for the values of i and j.

For brevity, a memory configuration characterized by (ℓ, m), is a particular realization of the L-M memory organization discussed above, where the number of lines, $\ell = 2^b$ and the number of modules per line $m = 2^{n-b}$. For example, if $b = 3$, $n = 5$, then $\ell = 8$ and $m = 4$. Hence we have a memory configuration of $(\ell, m) = (8, 4)$.

2.3 Memory Request Scheduling

Each pipelined processor issues one memory request every STU and p parallel processors issue p simultaneous requests each STU. Of these p parallel requests, some of them might address the same line resulting in a conflict. Even when all p simultaneous requests address distinct lines, conflict can still result if a request addresses a line or module which is still executing a previous request. Such a line or module which is executing a request at time t is said to be busy or active at time t. If a line or module is not busy, it is said to be idle or inactive.

Definition 2.3.1 A memory request collision is said to occur when a memory request attempts to access a busy line or module, or when at least two simultaneous requests attempt to access the same line. □

When more than one request attempts to access the same line simultaneously, a multiple access line collision occurs. When a request attempts to access a busy line, a line collision occurs. Similarly, a module collision occurs when a request attempts to access a busy module.

Definition 2.3.2 The status of Module $M_{i,j}$ at time t is L_i busy or idle at t, and $M_{i,j}$ busy or idle at time t.

The status of a memory module addressed by request is required to determine the outcome of the request. A request can access module $M_{i,j}$ at t if and only if $M_{i,j}$ and its line L_i, are both idle at t. Hence a request is rejected if it addresses a busy line or module. However, if one or more simultaneous requests refer to idle modules on the same idle line, one of these is accepted and the others rejected. One method of handling rejected requests is to recycle them through their corresponding processor segments for one memory cycle and resubmit them as new memory requests one instruction cycle later. During the recycling of each rejected request, a flag is set in the process of the rejected request to deactivate execution of that process until the request is accepted, whereupon the flag is reset and execution is reactivated.

Following an initiation of a memory operation at time t on line L_i and module $M_{i,j}$, L_i is busy with respect to other requests at time $t, t + 1, \ldots, t + a - 1$. Similarly, $M_{i,j}$ is busy with respect to other requests at time $t, t + 1, \ldots, t + c - 1$. Hence L_i and $M_{i,j}$ remain busy in the interval $\langle t, t + a \rangle$ and $\langle t, t + c \rangle$ respectively.

When a multiple access line collision occurs on one line, only one of the requests may be accepted. A request is termed an acceptable request if it addresses an idle module on an idle line. If there is only one such request for a line, the request will be accepted. However, when there is more than one such acceptable request, one of them is accepted arbitrarily and the others rejected. However, it should be pointed out that the acceptance of a request may depend on whether the busy module rejection is by the module itself or is centralized. In practice, as illustrated by the simulation model discussed in Chapter 4, a priority scheme may be adopted to select one of the acceptable requests to be accepted. One such priority scheme assigns a distinct priority to each processor so that any request issued by a particular processor has, associated with it, the processors priority number. Another scheme selects one of the acceptable requests according to a round-robin processor priority assignment.

In summary, there are basically three different types of stumbling blocks which can deter a request from being accepted by the memory system. A request may be rejected due to

(1) multiple access line collision, which may occur only if $p > 1$,

(2) line collision, which may occur only if $a > 1$, and

(3) module collision due to a busy module on an idle line, which may occur if $c > a$.

2.4 Processor-Memory Interconnection

On the arrival of p simultaneous requests at time t, the following

situations must be checked before any request is accepted.

(i) The status of each module referenced by each request.

(ii) Multiple access line collisions on lines associated with referenced modules.

When these items have been checked and all conflicts resolved, the accepted requests must be routed to the appropriate lines and modules. Basically, functional units are required to (1) maintain status of currently busy lines and modules; (2) resolve multiple access line collisions; and (3) successfully route accepted addresses to referenced lines.

Details of such functional units pose a major problem in the design of real parallel-pipelined computer systems. There exist a wide variety of possible implementations of such devices and a particular choice depends on the designer's objectives.

The functional unit required to store and update currently busy module status may also be required to accept or reject incoming requests. Such a unit would perform a module and line busy check for each of the incoming requests. Requests to idle modules on idle lines would then be steered through a p to ℓ crossbar switch which would arbitrate requests to the same line. The crossbar dimensions are hopefully significantly smaller than the p to N crossbar required by previous memory system organizations (with $s = 1$ and $\ell = N$).

Two possible implementations of busy check hardware are readily apparent. Each involves a small memory to store module busy status and another for line busy status. The first scheme employs N shift registers of $c - 1$ bits each and ℓ shift registers of $a - 1$ bits each

as the two memories. These are shifted right once per STU with 1's introduced on the left. The least significant bits of the p requested locations are read. An addressed module or line is busy if 0 is read from either register. In such a case, the corresponding request is rejected. If a forwarded request is accepted by the crossbar switch, the corresponding module and line shift registers are cleared to 0.

An alternative implementation employs two multi-access content addressable memories of size $p(c - 1)$ words by $\log_2 (N + 1)$ bits to store busy module addresses or "blanks" and $p(a - 1)$ words by $\log_2 (\ell + 1)$ bits to store busy line addresses or blanks. Notice that if $a = c$, only one multi-access CAM is required and it is used to store the addresses of busy lines. In each memory, p words are addressed by content each STU. A bit implies that the line or module referenced is busy. The memories are divided into successive blocks of p locations. One block, selected in a round-robin fashion, is overwritten each STU with addresses of new requests or "blanks" if some requests were not accepted.

One might be concerned that the functional units appear to be infinitely fast. This assumption is adequate for the model and simplifies the discussion. In practice, requests should arrive early, e.g., at t in Figure 2.2.4. The functional unit would then be active during $<t, t + 1>$ and the line and module activity would be unchanged. Alternatively, s may be increased for the purpose of adding STUs for memory management. As will be seen, performance is not a function of s and is not thus affected by the time required by the accept/reject logic, provided an adequately pipelined processor is used.

3. PERFORMANCE ANALYSIS OF L-M MEMORY ORGANIZATION

3.1 Introduction

In a parallel-pipelined processor of order (s, p), p simultaneous memory requests can be issued to the memory system every segment time unit. For analytical purposes, it is assumed that the addresses of the requests are independent and uniformly distributed among the N identical memory modules. This assumption yields a conservative estimation of performance. If in some instruction cycles memory is not referenced, the performance will be higher than indicated since there would be less conflict.

Although the sp instruction streams are independent, a single instruction stream is often vectoral, referencing distinct lines in sequence. However, since $s \geq c$, a memory module which is executing a previous request from an instruction stream would have completed its execution when the next request from the same instruction stream arrives. Hence it would appear that since there is no execution overlap between instructions of the same stream, program locality will not affect the performance significantly. As will be seen in the analysis to be presented, performance variation is largely due to the module characteristics (a, c), the memory configuration (ℓ, m), and the processor characteristics p and τ.

In order to refine the randomness assumption for the analytical model, it will also be presumed that rejected memory requests due to

line or module collisions are discarded. The discarding of rejected requests justifies the assumption that the addresses of the requests issued are independent. This justification is tested in the simulation model. In a practical case, one method of handling such collisions is to cause the process with a rejected memory request to make a non-computing pass through the processor segments for one instruction cycle and to resubmit the request the next cycle. In such cases, the process is blocked until its request is accepted. Case studies of this and other practical cases will be discussed fully in Chapter 4.

The performance analysis of the memory organization is carried out using discrete Markov models. The Markov models were used to analyze the effect of pipelining in the system. Hence, the models discussed in the following sections assume that $p = 1$. The effect of parallelism, $p > 1$, on the performance will be discussed in section 3.5.

The analytical models belong to a class of simple homogeneous discrete Markov chains with a finite number of mutually exclusive and exhaustive states [23]. For performance evaluation of given module characteristics (a, c), and memory configuration (ℓ, m), analysis is oriented toward developing the "probability of acceptance": the probability of being in any one of a certain class of states.

One inference that can be made directly from the assumption, involving uniformly distributed independent addresses, is that the probability of a request addressing any module is $1/N$. Similarly, since lines are identical and independent, the probability of a request addressing a line is $1/\ell$.

3.2 State Diagrams for p = 1

First we develop the state space of the memory system for the case p = 1. In this section, we assume c > 1 since c = 1 is degenerate and trivial.

Definition 3.2.1 A module state at time t is

= { } = ∅ (null), if the module is idle at t,

= {r}, if the module is busy at t because it accepted a request r STUs ago, where r is an integer such that $1 \leq r \leq c - 1$. □

Observe that the state of a module at t which accepted a request c STUs ago is ∅. Recall that since the memory cycle is c, the module was busy in the interval <t - c, t>. Since there are m modules on a line, the states of all m modules on the line represent the state of the line.

Definition 3.2.2 A line state, λ(t), at time t is the set union of all module states at time t for all modules on the line in question. For convenience, the line state is enclosed in "(" and ")". □

Notice that only nonnull states of modules on the line are required to specify the state of the line. Note that the line state only identifies whether some module on the line is in each state, and not which particular modules are in which state. Specific module information is not needed due to the uniformity and independence of accessing assumption. Furthermore, there can be at most one module on a line in any nonnull state, thus there are no repeated nonnull module states and a simple set

union of module states gives the line state. If there is more than one busy module on the line at time t, the module states are separated by a comma. For instance, consider the line state of a line at t which has two busy modules on it, one of which accepted a request one STU ago and the other, three STUs ago. The line state at t for this line will be denoted by (1, 3). For convenience, the module states of a line state will be listed in ascending order of busy time. Hence (3, 1) will be written as (1, 3). Moreover, if all modules on a line are idle at t then the line state is denoted by the empty set, ( ) = $\emptyset$.

Since there are ℓ lines in the system, the line states of all ℓ lines will specify the state of the system.

Definition 3.2.3 The system state, $\sigma(t)$, at time t is the unordered set of all nonempty line states at time t for all lines in the system. The system state is enclosed in "[" and "]". □

If the line state for all lines is $\emptyset$, then the system state is denoted by the set consisting of the empty set [( )] = [$\emptyset$]. As an example of system state, consider the system with two nonempty line states (1, 3) and (2). The system state will be denoted by [(1, 3)(2)]. For convenience, the line states will be listed in ascending order of the first module state in each line state.

Given the state of the system at t, it is necessary to determine the system state at time t + 1. To make this determination, it is necessary to understand the change of module states with time. Given the module state of a module at time t, the module state at time t + 1 can

be evaluated if it is known whether a request is made to and accepted by that module.

Definition 3.2.4 The next or successor state of a state is the state at time t + 1 given the state and input at time t. □

Hence for a given module state, the next module state can be obtained as follows.

Definition 3.2.5 Given that the state of a module at t is Ø, the next module state (at t + 1) is

= {1}, if a request which addressed the module at t was accepted, or

= Ø otherwise; i.e., either no request addressed the module at t, or a request which addressed the module at t was rejected due to a line collision. □

Therefore, a module remains in the null state unless it accepts a request at time t whereupon it will become busy and remain busy during the interval $<t, t + c>$. For a busy module, the next state can now be evaluated as follows:

Definition 3.2.6 Given that the state of a module at t is {r}, where r is an integer such that $1 \leq r \leq c - 1$, the next module state is {r + 1} if $r < c - 1$ and Ø if $r = c - 1$. □

Once a module accepts a request, it goes through the module states {1}, {2}, ...{c-1}, ∅. Hence the maximum utilization of a module is one accepted request per c STUs. Only a module in the module state ∅ can accept a request.

It need not be known whether a request addressed a busy module in order to evaluate the next state of the module. Any request made to a busy module is rejected.

The line state as defined does not really explain the constraining factor in the formation of a line state. The constraining factor is the module characteristics (a, c), which implies that a line which accepts a request will remain busy for a STUs. Hence no two busy modules on a line will have their module states less than a STUs apart. This leads to the formal definition of a realizable line state.

Definition 3.2.7 A line state $\lambda(t)$, is said to be a realizable line state for module characteristics (a, c) if $\lambda(t) \subseteq (1, 2, \ldots, c-1)$ and

1. it is the null state, ∅, or
2. it consists of one element, r, or
3. it consists of two or more elements such that for any r_i, $r_j \in \lambda(t)$, $|r_j - r_i| \geq a$. □

As an illustration, the following are all the realizable line states for the module characteristics (a, c) = (2, 6): ∅, (1), (2), (3), (4), (5), (1, 3), (1, 4), (1, 5), (2, 4), (2, 5), (3, 5) and (1, 3, 5).

Determining the next line state is not as straightforward as determining the next module state. Hence some definitions are needed here to clarify the presentation.

A line state is an acceptance state if it contains the element r = 1. A particular line state enters an acceptance state one STU after it was addressed by an accepted request. Line states which are not acceptance states are called nonacceptance states. Similarly, a system state is an acceptance state if it contains an acceptance line state, otherwise it is a rejection state.

There are at most two possible state transitions from a particular line state, $\lambda(t)$: one is to an acceptance state and the other to a nonacceptance state. The system state $\sigma(t)$ which is not an empty state can make at least two possible state transitions, namely, to the rejection state and to one of its possibly many acceptance states. Since p = 1, it is obvious that the empty system state [Ø] can only make a transition to the acceptance system state [(1)].

Two state transitions, namely, generative and regenerative, are introduced to aid in generating the state transition graph. Generative transitions generate new states, whereas regenerative transitions generate states which have already appeared in the state transition graphs. Hence a transition is regenerative for c > 1, if its source state contains the module state r = c - 1, otherwise, it is generative. For generative and regenerative transitions, the source states are called generator and regenerator states respectively. Note for c = 1, the null state is the only state of the system, and the definitions above do not apply.

For example, if a request references a line with no busy modules on it, the request is accepted and at the next time instant, a module on that line would have been busy for one STU. Hence a generative transition occurs from the null line state, $\emptyset$, to the acceptance line state, (1). Similarly, if no request referenced that line, all modules on that line will be idle at the next time instant. Hence a regenerative transition occurs from the null line state, $\emptyset$, to the nonacceptance line state, $\emptyset$.

Definition 3.2.8 If $\exists\ r \in \lambda(t)$ such that $1 \leq r < a$, then $\lambda(t)$ is a busy line state, otherwise it is an idle line state. □

For a busy line state, the line is busy and will only make a transition to its nonacceptance successor state whether or not a request addresses the line. Notice that if $a = 1$, there are no busy line states.

In order to develop the Markov model required to analyze the memory conflict problem on a line, the line state space is investigated. However, we need to know the probability of transition from one state to the other in order to compute the probability of being in either of two classes of states called acceptance and nonacceptance states. The cardinality of states is useful in obtaining the transition probabilities.

Definition 3.2.9 The set of all realizable line states for module characteristics (a, c) is $\Lambda(a, c)$. Similarly, the set of all realizable idle line states for module characteristics (a, c) is $\Lambda_I(a, c)$. □

Hence, $\Lambda_I(a, c) = \{ \lambda \mid \lambda \in \Lambda(a, c)$ and if $r \in \lambda$ then $r \geq a \} \subseteq \Lambda(a, c)$. It would be appropriate to represent the set of all realizable busy line states for module characteristics (a, c) by $\Lambda_B(a, c) = \Lambda(a, c) - \Lambda_I(a, c)$. Notice that $\Lambda_B(a, c) \cap \Lambda_I(a, c) = \emptyset$. Recalling that the system state at time t is composed of all the line states at time t in the system, each system state consists of two disjoint subsets of line states namely, the set of idle line states and busy line states at t.

Definition 3.2.10 The number of elements or the cardinality of a line state $\lambda(t)$, $|\lambda(t)|$, is the number of nonnull module states in the line state. Similarly, the cardinality of the system state $\sigma(t)$, $|\sigma(t)|$, is the number of nonempty line states in $\sigma(t)$. □

For example, if $\sigma(t) = [\emptyset]$, $|\sigma(t)| = 0$ and $|[(1, 3)(5)]| = 2$.

Recall that when a request references an idle module on an idle line, the request is accepted, causing the line state to make a transition to an acceptance line state. Similarly, a line state makes a transition to a nonacceptance line state if no request referenced the line or a request which referenced the line was rejected. The following definitions formalize the above description.

Definition 3.2.11 A line acceptance function, f_a, maps the idle line state $\lambda(t) \in \Lambda_I(a, c)$ to its next acceptance line state, $\lambda(t + 1) \in \Lambda(a, c)$, i.e., $f_a(\lambda(t)) = \lambda(t + 1)$. Hence,

$$f_a: \Lambda_I(a, c) \rightarrow \Lambda(a, c).$$ □

Notice that a busy line state does not have a next acceptance line state.

Definition 3.2.12 A line nonacceptance function, f_n, maps the line state $\lambda(t) \in \Lambda(a, c)$ to its next nonacceptance line state, $\lambda(t + 1) \in \Lambda(a, c)$, i.e., $f_n(\lambda(t)) = \lambda(t + 1)$. Hence

$$f_n: \Lambda(a, c) \rightarrow \Lambda(a, c).$$ □

The following theorems in this section are valid for the p = 1 case.

Theorem 3.2.1 If no request is accepted at time t by a line in state $\lambda(t)$, the next state of that line is the nonacceptance state

$$f_n(\lambda(t)) = \lambda(t + 1) = (x \mid x - 1 \in \lambda(t) \text{ and } x < c).$$

Proof We can apply definitions 3.2.5 and 3.2.6 directly to each element of the line state, $\lambda(t)$ to obtain $f_n(\lambda(t))$. Since no request is accepted by the line at t, the module state associated with each module on the line will make a transition to its next module state. Hence, each module in the null module state, $\emptyset \in \lambda(t)$ makes a transition to $\emptyset$ and each module in module state $\{r\}$, where $r \in \lambda(t)$ and $1 \leq r \leq c - 1$, makes a transition to $\{r + 1\}$ if $r < c - 1$ and to $\emptyset$ if $r = c - 1$. By substituting x for r + 1, the theorem follows. □

Corollary 3.2.1.1 For a transition from line state $\lambda(t)$ to its nonacceptance successor line state $\lambda(t + 1)$

$$|\lambda(t + 1)| = |\lambda(t)| - 1, \text{ if } c - 1 \in \lambda(t)$$
$$= |\lambda(t)|, \text{ otherwise.}$$

Proof If $\forall r \in \lambda(t)$, $r < c - 1$, then each element $r \in \lambda(t)$ is mapped to $r + 1 \in \lambda(t + 1)$. Hence $|\lambda(t + 1)| = |\lambda(t)|$. However, if $\exists r \in \lambda(t)$ such that $r = c - 1$, then the element $c - 1 \in \lambda(t)$ is not mapped to an element of $\lambda(t + 1)$. Hence, $|\lambda(t + 1)| = |\lambda(t)| - 1$ if $c - 1 \in \lambda(t)$. □

Notice that if $c - 1 \in \lambda(t)$, then the transition from $\lambda(t)$ to $f_n(\lambda(t))$ is regenerative, otherwise it is generative.

The following theorem is used to evaluate the next acceptance line state of an idle line state.

Theorem 3.2.2 If a request is accepted at time t by a line in state $\lambda(t)$, the next state of that line is

$$f_a(\lambda(t)) = \lambda(t + 1) = (1) \cup (x \mid x - 1 \in \lambda(t) \text{ and } x < c).$$

Proof A request is accepted by the line if the line is idle and a request references an idle module on the line. The next module state of the referenced idle module on the idle line is $\{1\}$. Concurrently, all other modules on the line make transitions to their respective next states. That is, the set of next module states from theorem 3.2.1 is $(x \mid x - 1 \in \lambda(t) \text{ and } x < c)$. Hence the next state of the line is $\lambda(t + 1) = (1) \cup f_n(\lambda(t))$. □

Hence if $\lambda(t)$ is an idle line state, $f_a(\lambda(t)) = (1) \cup f_n(\lambda(t))$.

Corollary 3.2.2.1 For a transition from idle line state $\lambda(t)$ to its acceptance successor line state $\lambda(t + 1)$,

$$|\lambda(t+1)| = |\lambda(t)|, \quad \text{if } c-1 \in \lambda(t)$$
$$= |\lambda(t)| + 1, \text{ otherwise}$$

□

This is obvious from theorem 3.2.2 and corollary 3.2.1.1. Given a line state, the next line state can therefore be evaluated from the two theorems above.

A request can address only one line at a given time instant. If the request is accepted on the addressed line, only its corresponding line state will be transformed to its next acceptance line state, while all other line states of the system state will be transformed into their corresponding next nonacceptance states. For example, if (a, c) = (2, 4), a request which addresses a line represented by the implicit idle line state Ø in the system state [(1) (2)] is accepted resulting in the next acceptance line state $f_a(\emptyset) = (1)$. The other line states, (1) and (2) make transitions to their next nonacceptance states, $f_n((1)) = (2)$ and $f_n((2)) = (3)$ respectively. Hence the resulting next system state is [(1) (2) (3)]. However, if the request is made instead to an idle module on the idle line represented by (2), it is accepted since 2 = a and the resulting next acceptance system state is [(1, 3) (2)]. Furthermore, if the request is made to the busy line represented by (1), it is rejected and the next rejection system state is [(2) (3)]. With these illustrations it can be seen that there may be multiple next acceptance system states. On the other hand, for any given system state, with the exception of the null system state, there exists only one next rejection system state. The null system state generates only the acceptance state [(1)], since any request which arrives will be accepted by an idle memory system.

In order to develop the Markov model for the memory conflict problem in the memory system for p = 1, the system state space is investigated. The system state space can be obtained systematically by generating the successor system states from a present state, given that a request was either rejected by the system or it was accepted by a particular idle line in the system.

Definition 3.2.13 S(a, c) is the set of all system states for given module characteristics (a, c). □

In systematizing the generation of all the system states for module characteristics (a, c), system acceptance and rejection functions similar to line acceptance and nonacceptance functions will be defined. Let $\sigma_\lambda(t + 1)$ represent the acceptance successor system state of the system state $\sigma(t)$ which accepted a request on an idle line with at least one idle module, represented by the idle line state $\lambda(t) \in \sigma(t)$. Assume for the moment that there are at least a lines in the system. Then, S(a, c) will contain a system state which has an idle line state which represents an idle line with at least one idle module.

Definition 3.2.14 A system acceptance function,

$$_\lambda g_a : S(a, c) \rightarrow S(a, c),$$

such that the request which is accepted on the idle line represented by the idle line state $\lambda \in \sigma(t)$ causes a transition from system state $\sigma(t)$ to next acceptance system state $\sigma_\lambda(t + 1)$. Hence $_\lambda g_a(\sigma(t)) = \sigma_\lambda(t + 1)$. □

Notice that $f_a(\lambda) \in \sigma_\lambda(t + 1)$. If a request is made to an empty system, the request will always be accepted. Hence the next system state of $[\emptyset]$ is $[(1)]$. However, for $p = 1$, one request arrives every STU, hence, an empty system cannot cause a rejection. Therefore, the memory system may reject a request only if the system state is nonnull. Let $\sigma_n(t + 1)$ represent the rejection successor system state of the system state $\sigma(t)$.

Definition 3.2.15 A system rejection function

$$g_r: S(a, c) - \{[\emptyset]\} \rightarrow S(a, c)$$

maps the nonnull system state $\sigma(t) \in S(a, c)$ to its next rejection state $\sigma_n(t + 1) \in S(a, c)$. Hence, $g_r(\sigma(t)) = \sigma_n(t + 1)$. □

Hence ${}_\lambda g_a$ and g_r are the state transformations corresponding to an acceptance on the idle line represented by the idle line state λ and a rejection by the system respectively.

Theorem 3.2.3 If a request is rejected at time t by the system in state $\sigma(t)$, the next state of the system is the rejection system state

$$\sigma_n(t + 1) = [\lambda_n \mid \lambda_n = f_n(\lambda), \lambda \in \sigma(t)].$$

Proof The result is directly obtained by application of the line non-acceptance function to each line state in the system state $\sigma(t)$, since a system rejection implies nonacceptance on all lines in the system. □

As an illustration, consider the system state $[(1)(2)]$ for module characteristics $(a, c) = (2, 4)$. The next rejection system state

$\sigma_n(t + 1) = [f_n(1)\ f_n(2)] = [(2)(3)]$. The next rejection system state for $[(2)(3)]$ is $[(3)]$.

For $a = c = 1$, the only state of the system is the system state $\sigma(t) = [\emptyset]$. In this case, the next state of the system is $\sigma(t + 1) = [\emptyset]$. Hence $|\sigma(t + 1)| = |\sigma(t)|$. Notice that for $p = 1$, there is always acceptance when $a = c = 1$. However, let us evaluate the cardinality of the next system state when a rejection occurs, assuming that $c > 1$, and $p = 1$.

Corollary 3.2.3.1 For the generative transition from a nonempty system state $\sigma(t)$ to its next rejection system state $\sigma_n(t + 1)$,

$$|\sigma_n(t + 1)| = |\sigma(t)|$$

Proof If $\sigma(t)$ is a system generator, then each nonempty line state $\lambda \in \sigma(t)$ is a line generator. Hence, $f_n(\lambda) \neq \emptyset$. Therefore, the system rejection function, g_r, induces a one-to-one onto mapping of the elements of $\sigma(t)$ to those of $\sigma_n(t + 1)$. Hence $|\sigma_n(t + 1)| = |\sigma(t)|$. □

Corollary 3.2.3.2 For regenerative transition from system state $\sigma(t)$ to its next rejection system state $\sigma_n(t + 1)$,

1. $|\sigma_n(t + 1)| = |\sigma(t)|$, if $\exists\ \lambda(t) \in \sigma(t) \mid c - 1 \in \lambda(t)$ and $|\lambda(t)| > 1$.
2. $|\sigma_n(t + 1)| = |\sigma(t)| - 1$, if $\exists\ \lambda(t) \in \sigma(t) \mid c - 1 \in \lambda(t)$ and $|\lambda(t)| = 1$.

Proof If $\exists\ \lambda \in \sigma(t) \mid c - 1 \in \lambda$ and $|\lambda| > 1$, then $f_n(\lambda) \neq \emptyset$. Since when $p = 1$, there is at most one line regenerator in a system regenerator, $|\sigma_n(t + 1)| = |\sigma(t)|$. However, if $\exists\ \lambda \in \sigma(t) \mid c - 1 \in \lambda$ and

$|\lambda| = 1$, then $f_n(\lambda) = \emptyset$. Hence, $|\sigma_n(t + 1)| = |\sigma(t)| - 1$. □

For example, if $(a, c) = (2, 4)$ and $\sigma(t) = [(1, 3)(2)]$, the next rejection state is $\sigma_n(t + 1) = [(2)(3)]$ and $|\sigma_n(t + 1)| = |\sigma(t)| = 2$. Notice that the line regenerator, $(1, 3) \in \sigma(t)$, is such that $|(1, 3)| = 2$. Suppose $\sigma(t) = [(1)(2)(3)]$, the next rejection state is $\sigma_n(t + 1) = [(2)(3)]$ and $|\sigma_n(t + 1)| = |\sigma(t)| - 1 = 2$. Notice that in this case, the only line regenerator in $\sigma(t)$ is the line state (3).

Theorem 3.2.4 If a request is accepted at time t on an idle line represented by the idle line state $\lambda \in \sigma(t)$, the next state of the system is the acceptance state,

$$\sigma_\lambda(t + 1) = [\lambda_n \mid \lambda_n = f_n(\lambda_i), \lambda_i \in \sigma(t) \text{ and } \lambda_i \neq \lambda] \cup [f_a(\lambda)].$$

Proof An acceptance on the line represented by λ generates the line state $f_a(\lambda)$. All other line states in $\sigma(t)$ make their respective transitions to their next nonacceptance line states. The set of these next nonacceptance states is $[\lambda_n \mid \lambda_n = f_n(\lambda_i), \lambda_i \in \sigma(t) \text{ and } \lambda_i \neq \lambda]$. Therefore, $\sigma_\lambda(t + 1) = [\lambda_n \mid \lambda_n = f_n(\lambda_i), \lambda_i \in \sigma(t) \text{ and } \lambda_i \neq \lambda] \cup [f_a(\lambda)]$. □

Notice that if $\sigma_n(t + 1)$ is the next rejection system state of $\sigma(t)$, then $[\lambda_n \mid \lambda_n = f_n(\lambda_i), \lambda_i \in \sigma(t) \text{ and } \lambda_i \neq \lambda] = \sigma_n(t + 1) - [f_n(\lambda)]$. Hence, $\sigma_\lambda(t + 1) = (\sigma_n(t + 1) - [f_n(\lambda)]) \cup [f_a(\lambda)]$.

Corollary 3.2.4.1 If the idle line state $\lambda = \emptyset$ or $\lambda \in \sigma(t)$ is such that $c - 1 \in \lambda$ and $|\lambda| = 1$,

$|\sigma_\lambda(t+1)| = |\sigma_n(t+1)| + 1$, where $\sigma_n(t+1)$ is the next rejection system state of $\sigma(t)$.

Proof From the above theorem, $|\sigma_\lambda(t+1)| = |\sigma_n(t+1)| - |[f_n(\lambda)]| + |[f_a(\lambda)]|$. If $\lambda = \emptyset$ or $\lambda \in \sigma(t)$ is such that $c-1 \in \lambda$ and $|\lambda| = 1$, $|[f_n(\lambda)]| = 0$. Howev er, $|[f_a(\lambda)]| = 1$ for any idle line state $\lambda \in \sigma(t)$, hence the result follows. □

Corollary 3.2.4.2 If $\lambda \neq \emptyset$ and is a generator, or $\lambda \in \sigma(t)$ is such that $c-1 \in \lambda$ and $|\lambda| > 1$,

$$|\sigma_\lambda(t+1)| = |\sigma_n(t+1)| .$$ □

The proof of this is similar to that of corollary 3.2.4.1. However, in this case, $|[f_n(\lambda)]| = 1$.

For example if (a, c) = (2, 4), let $\sigma(t) = [(1)\ (2)]$. The next rejection system state of [(1) (2)] is $\sigma_n(t+1) = [(2)\ (3)]$. For this system state there are two distinct idle line states namely, $\lambda_0 = \emptyset$ and $\lambda_1 = (2)$. Hence the exhaustive next acceptance system states of [(1)(2)] are $\sigma_\emptyset(t+1) = [f_a(\emptyset)] \cup (\sigma_n(t+1) - [f_n(\emptyset)]) = [(1)] \cup ([(2)(3)] - [\emptyset]) = [(1)(2)(3)]$ and $\sigma_{(2)}(t+1) = [(1,3)] \cup ([(2)(3)] - [(3)]) = [(1,3)(2)]$. Tables 3.2.1 and 3.2.2 summarize the next state and its cardinality as a function of the present state for line states and system states respectively.

Hence from the last two theorems, it is obvious that if a state $\sigma(t)$ can make a transition to $\sigma_n(t+1) = g_r(\sigma)$, it can also make a transition to $\sigma_\lambda(t+1) = {}_\lambda g_a(\sigma)$, for each idle line state $\lambda \in \sigma(t)$. It is interesting to note that for generative transitions, there is only one state in the system which makes a transition to a particular nonnull rejection system state.

TABLE 3.2.1 Next line state (NLS) and its cardinality as a function of present line state (PLS)

PLS $\lambda(t)$	Outcome (function)	NLS $\lambda(t+1)$	Cardinality $\lvert\lambda(t+1)\rvert$
$\lambda(t)$	nonacceptance f_n	$f_n(\lambda(t)) =$ $(x \mid x\text{-}1 \in \lambda(t)$ and $x < c)$	$\lvert\lambda(t)\rvert$-1, if c-1 $\in \lambda(t)$. $\lvert\lambda(t)\rvert$, otherwise
$\lambda(t)$	acceptance f_a	$f_a(\lambda(t)) =$ $(1) \cup f_n(\lambda(t))$	$\lvert\lambda(t)\rvert$, if c-1 $\in \lambda(t)$ $\lvert\lambda(t)\rvert$ +1, otherwise

TABLE 3.2.2 Next system state (NSS) and its cardinality as a function of present system state (PSS).

PSS $\sigma(t)$	Outcome (function)	NSS $\sigma(t+1)$	Cardinality $\lvert\sigma(t+1)\rvert$
$\sigma(t) \neq [\emptyset]$	rejection g_r	$g_r(\sigma(t)) =$ $\sigma_n(t+1) =$ $[\lambda_n \mid \lambda_n =$ $f_n(\lambda), \lambda \in \sigma(t)]$	$\lvert\sigma(t)\rvert$, if each $\lambda \in \sigma(t)$ is a line generator or, $\exists$ a line regenerator $\lambda \in \sigma(t) \mid \lvert\lambda\rvert > 1$. $\lvert\sigma(t)\rvert$-1, if $\exists$ a line regenerator $\lambda \in \sigma(t) \mid \lvert\lambda\rvert = 1$.
$\sigma(t)$	acceptance on line state, $\lambda \in \sigma(t)$. λ^{g_a}	$\lambda^{g_a}(\sigma(t)) =$ $\sigma_\lambda(t+1) =$ $[\lambda_n \mid \lambda_n =$ $f_n(\lambda_i), \lambda_i \in \sigma(t)$ and $\lambda_i \neq \lambda] \cup$ $[f_a(\lambda)]$	$\lvert\sigma_n(t+1)\rvert$+1, if $\lambda = \emptyset$ or λ is a line regenerator $\mid \lvert\lambda\rvert = 1$. $\lvert\sigma_n(t+1)\rvert$, if λ is a line generator $\lvert\lambda \neq \emptyset$ or, λ is a line regenerator $\mid \lvert\lambda\rvert > 1$.

That is, there are no two rejection states which are successors of the same state. However, there may exist several acceptance states in the system which are successors of the same state. As will be seen later for regenerative transitions, more than one regenerative system state may make a transition to the same system state.

The set of all system states, S(a, c) for given (a, c) can be generated systematically by the application of the system acceptance and rejection functions. In addition a system state graph can be used to display the next system state function. In this graph, there is one node for each system state in S(a, c) and one edge leaving each node for each possible outcome of a request. The paths on the graph show the state changes experienced by the system for all request sequences. In particular, the directed edge indicates a transition from one state at time t to another at time t + 1. An algorithm to generate S(a, c) and obtain a system state graph starting from the null system state will now be developed.

Let $V_o = \{[\emptyset]\}$. Generally, V_n is the set of system states generated by one-step transitions from the states in V_{n-1}. For example, V_1 is the set of system states generated by the null system state, $[\emptyset]$ which is the only element in V_o. The set of rejection states generated by the states in V_{n-1} is called R_n, where $R_n = \{\sigma_j | \sigma_j = g_r(\sigma_i), \sigma_i \in V_{n-1}\}$. Similarly, the set of acceptance states generated by the states in V_{n-1} is called A_n, where $A_n = \{\sigma_\lambda | \sigma_\lambda = {}_\lambda g_a(\sigma)$, for some idle line state $\lambda \in \sigma$ and $\sigma \in V_{n-1}\}$. Notice that the null line state, $\emptyset \in \sigma$, is an idle line state. Then $V_n = A_n \cup R_n$. Notice that R_1 is an empty set since $g_r([\emptyset])$ does not exist. Hence $R_1 = \emptyset$ and $V_1 = A_1 = \{[(1)]\}$. Similarly, $A_o = \emptyset$,

while $R_o = \{[(\emptyset)]\} = V_o$. Notice that in general $\sigma \in V_n$ if and only if $\exists\ \lambda \in \sigma$ such that $n \in \lambda$, for $1 \leq n \leq c - 1$. It will be shown that V_{c-1} is the set of all regenerative system states for given (a, c). However, let us introduce some definitions and lemmas to aid in the development.

Definition 3.2.16 For generative transitions, the inverse line function, f^{-1}, maps the line state, $\lambda(t) \in \Lambda(a, c)$ to its predecessor line state $\lambda(t - 1) \in \Lambda(a, c)$, i.e., $f^{-1}(\lambda(t)) = \lambda(t - 1)$. Hence, f^{-1}: $\Lambda(a, c) \rightarrow \Lambda(a, c)$.

□

The inverse line function is defined for generative transitions only. In this case, the predecessor state is unique. That is, if all regenerative transitions are ignored, there exists one and only one predecessor state for each line state in $\Lambda(a, c)$. Notice that f^{-1} is neither onto nor one-to-one mapping since no state is mapped to regenerative line states.

Lemma 3.2.1 For generative transitions, the predecessor state of a line state $\lambda(t)$ is $\lambda(t-1) = f^{-1}(\lambda(t)) = (x | x+1 \in \lambda(t) \text{ and } x > 0)$.

Proof The proof of this is straightforward from theorems 3.2.1 and 3.2.2, since $\lambda(t) = f_n(\lambda(t-1))$ or $\lambda(t) = f_a(\lambda(t-1))$.

□

Hence for the (a, c) = (2, 4) example, if $\lambda(t) = (1,3)$, $f^{-1}(\lambda(t)) = (2)$. Similarly, if $\lambda(t) = (2)$, $f^{-1}(\lambda(t)) = (1)$.

Similarly, for generative transitions, we can determine the predecessor state of a nonnull system state from the following definition and lemma.

Definition 3.2.17 For generative transitions, the inverse system function, g^{-1}, maps the nonempty system state, $\sigma(t) \in S(a, c)$, to its predecessor state, $\sigma(t-1) \in S(a, c)$, i.e., $g^{-1}(\sigma(t)) = \sigma(t-1)$. Hence g^{-1}: $S(a, c) - \{[(\emptyset)]\} \rightarrow S(a, c)$.

□

g^{-1} is neither onto nor one-to-one mapping since no state is mapped to regenerative system states.

Lemma 3.2.2 For generative transitions, the predecessor state of a nonempty system state $\sigma(t)$ is

$$\sigma(t-1) = g^{-1}(\sigma(t)) = [\lambda | \lambda = f^{-1}(\lambda_i),\ \lambda_i \in \sigma(t)].$$

Proof The proof of this can be obtained directly from theorems 3.2.3 and 3.2.4 and Lemma 3.2.1, since $\sigma(t) = g_r(\sigma(t-1)$ or $\sigma(t) = {}_{\lambda}g_a(\sigma(t-1))$, for some idle line state $\lambda \in \sigma(t-1)$.

□

For the (a, c) = (2, 4) example, if $\sigma(t) = [(3)]$ or $[(1, 3)(2)]$, then $g^{-1}(\sigma(t)) = [(2)]$ or $[(1)(2)]$ respectively. In general, we can denote the jth successive application of g^{-1} to σ as g^{-j} for $j \geq 0$. Hence $g^{-j}(\sigma) = g^{-1}(g^{-(j-1)}(\sigma))$. If $j = 0$, then $g^{-0}(\sigma) = \sigma$.

Theorem 3.2.5 V_{c-1} is the set of all regenerative system states for given (a, c).

Proof We know that V_{c-1} is the set of system states such that for every $\sigma \in V_{c-1}$, $\exists\ \lambda \in \sigma$ such that $c-1 \in \lambda$. If $\sigma \in V_{c-1}$ then $g^{-1}(\sigma) \in V_{c-2}$. If a state is regenerative then by definition it is in V_{c-1}. Now, if all the states in V_{c-1} are not regenerative then $\exists\ \sigma \in V_{c-1}$ such that σ does not make transitions to states already generated. That is, at least one of

$g_r(\sigma)$ and ${}_\lambda g_a(\sigma)$, for an idle line state $\lambda \in \sigma$, is not in $\bigcup_{i=0}^{c-1} V_i$. We know that if σ can make a transition to $g_r(\sigma)$, it can also make transitions to ${}_\lambda g_a(\sigma)$ for any idle line state $\lambda \in \sigma$. Hence it suffices to show that $g_r(\sigma) \in \bigcup_{i=0}^{c-1} V_i$, since if $g_r(\sigma) \in \bigcup_{i=0}^{c-1} V_i$, ${}_\lambda g_a \in \bigcup_{i=0}^{c-1} V_i$ for any idle state $\lambda \in \sigma$.

Since $\sigma(t) \in V_{c-1}$ contains the module state c-1, a transition from $\sigma(t)$ to $g_r(\sigma)$ at t+1 implies that the module whose module state is c-1 is idle at t + 1. Hence for σ, the module state, c - 1, does not contribute any state information at t+1. Therefore, for each $\sigma \in V_{c-1}$, remove the module state c - 1 from its regenerative line state. Let σ' represent the resulting system state. Then for nonnull σ', $g_r(\sigma) = g_r(\sigma')$. If $\sigma = [(c - 1)]$, then $\sigma' = [\emptyset]$ and $g_r(\sigma) = \sigma' = [\emptyset]$. Notice that the largest module state in nonnull σ' is less than c - 1. For the (a, c) = (2, 4) example, if $\sigma = [(1,3)(2)]$, then $\sigma' = [(1)(2)]$ and $g_r(\sigma) = g_r(\sigma')$ $= [(2)(3)]$. Let V'_{c-1} represent the set of σ' such that $\sigma \in V_{c-1}$. If we show that $\sigma' \in \bigcup_{i=0}^{c-2} V_i$, then $g_r(\sigma') \in \bigcup_{i=0}^{c-1} V_i$. By successive application of Lemma 3.2.2, $\exists$ j which will make $g^{-j}(\sigma) = [\emptyset]$, for some $j \leq c - 2$, since the largest element of any nonnull line state in σ' is less than c - 1. Hence it is always possible to make transitions from $[\emptyset]$ to σ' in j time instants. Hence $\sigma' \in \bigcup_{i=0}^{c-2} V_i$. Since all states in V_{c-2} make transitions to states in V_{c-1}, $g_r(\sigma') \in \bigcup_{i=0}^{c-1} V_i$. Therefore, $g_r(\sigma) \in \bigcup_{i=0}^{c-1} V_i$ and the theorem follows. □

Hence $V_0, V_1, \ldots, V_{c-1}$ generate all the system states while simultaneously creating all generative transitions. Thus $S(a, c) = \bigcup_{i=0}^{c-1} V_i$. In the process of forming V_c, all regenerative transitions are created.

This completes the formation of the system state graph.

Theorem 3.2.6 $V_c = S(a, c)$, if $\ell \geq a$ and $m > \left\lfloor \frac{c-1}{a} \right\rfloor$.

Proof From Theorem 3.2.5, we know that every system state $\sigma \in V_{c-1}$ always makes a transition to a state in S(a,c). We need to show that for every state $\sigma_i \in S(a,c)$, $\exists\ \sigma \in V_{c-1}$ which makes a transition to σ_i. States in S(a,c) can be partitioned into two disjoint subsets, namely, sets of rejection and acceptance states. That is, $S(a,c) = \bigcup_{i=0}^{c-1} R_i \cup \bigcup_{i=0}^{c-1} A_i$. We know that if $\sigma \in V_{c-1}$, σ makes a transition to $\sigma_n \in \bigcup_{i=0}^{c-1} R_i$ and also makes a transition to $\sigma_\lambda \in \bigcup_{i=0}^{c-1} A_i$, where $\sigma_\lambda = {}_\lambda g_a(\sigma)$, for each idle line state $\lambda \in \sigma$. Since $\ell \geq a$ and at most a-1 lines are busy, there is always an idle line and since $m > \left\lfloor \frac{c-1}{a} \right\rfloor$ there is always an idle module on an idle line. Also from the previous theorem, we know that if $\sigma \in V_{c-1}$, for every $\sigma_\lambda \in \bigcup_{i=0}^{c-1} A_i$, $\exists\ \sigma_n \in \bigcup_{i=0}^{c-1} R_i \mid \sigma_\lambda = (\sigma_n - [f_n(\lambda)]) \cup [f_a(\lambda)]$, for idle line state $\lambda \in \sigma$. Hence we only need to show that g_r: $V_{c-1} \rightarrow \bigcup_{i=0}^{c-1} R_i$ is an onto mapping in order to prove the theorem.

There are two kinds of system states in V_{c-1}, namely, states in which $\lambda_j \in \sigma \mid \lambda_j = (c-1)$, and states in which $\exists\ \lambda_j \in \sigma \mid c - 1 \in \lambda_j$ and $|\lambda_j| > 1$. Hence the former system state can be represented by $\sigma_A = [\lambda_1 \ldots \lambda_k\ \lambda_{k+1}]$, where $\lambda_1, \lambda_2, \ldots, \lambda_{k+1}$ are line states and $\lambda_{k+1} = (c - 1)$. The latter system state can be represented by $\sigma_B = [\lambda_1 \ldots \lambda_k]$, where $\lambda_1, \lambda_2, \ldots \lambda_k$ are line states and $\exists\ \lambda_j \in \sigma_B \mid c - 1 \in \lambda_j$ and $|\lambda_j| > 1$.

$\sigma_n \in \bigcup_{i=0}^{c-1} R_i$ can be represented by $\sigma_n = [\lambda_1' \ldots \lambda_k']$ where λ_1', λ_2', ..., λ_k' are rejection line states. A state $\sigma \in V_{c-1}$ can make a transition to σ_n if $\sigma_n = g_r(\sigma) = [\lambda_j' \mid \lambda_j' = f_n(\lambda_j), \lambda_j \in \sigma]$, from Theorem 3.2.3. It can be seen that if $\sigma = \sigma_A$ or σ_B the transition from $\sigma \in V_{c-1}$ to $\sigma_n \in \bigcup_{i=0}^{c-1} R_i$ may occur.

□

An example may be appropriate here. In order to obtain S(2, 4) and the system state graph for (a, c) = (2, 4), start with $V_0 = \{ [\emptyset] \}$. Then $V_1 = A_1 = \{ [(1)] \}$, where $[(1)] = {}_{\emptyset}g_a([\emptyset])$. Hence there is a directed edge from state $[\emptyset]$ to $[(1)]$ as shown in figure 3.2.1. The arc from the null system state to the acceptance state $[(1)]$ is identified by ${}_{\emptyset}g_a$. In general, arc labels in figure 3.2.1 are identified by g_r and ${}_{\lambda}g_a$ if the transition is due to a rejection and if the transition is due to an acceptance on the line represented by the idle line state λ respectively. $R_2 = \{ [(2)] \}$, where $[(2)] = g_r([(1)])$ and $A_2 = \{ [(1)(2)] \}$, where $[(1)(2)] = {}_{\emptyset}g_a([(1)])$. Hence $V_2 = A_2 \cup R_2 = \{ [(2)], [(1)(2)] \}$. Similarly, $R_3 = \{ [(3)], [(2)(3)] \}$ and $A_3 = \{ [(1)(3)], [(1, 3)], [(1)(2)(3)], [(1, 3)(2)] \}$. $V_3 = A_3 \cup R_3$. Notice that all elements of V_i, for $1 \leq i \leq c - 1$, contain a line state containing i. In particular, for $i = c - 1 = 3$, V_3 is the set of regenerative states. In creating all the regenerative transitions, notice that $R_4 = \bigcup_{i=0}^{3} R_i$ and $A_4 = \bigcup_{i=0}^{3} A_i$. Therefore, $V_4 = S(2, 4) = \bigcup_{i=0}^{3} V_i$.

In addition to the graphical illustration of state transformations, the system state graph aids in the performance analysis of the system. The performance of the system will be measured by the probability of accepting a request into the memory system. In order to determine the probability of acceptance, the probability of transition from one system state to the other should be known.

Definition 3.2.18 The probability of transition, p_{ij}, is the conditional probability of going from system state $\sigma_i(t)$, at time t, to its successor system state $\sigma_j(t + 1)$, at time t + 1. Rewriting this statement in probability notation, $p_{ij} = P(\sigma_j, t + 1 \mid \sigma_i, t)$, t is an integer. □

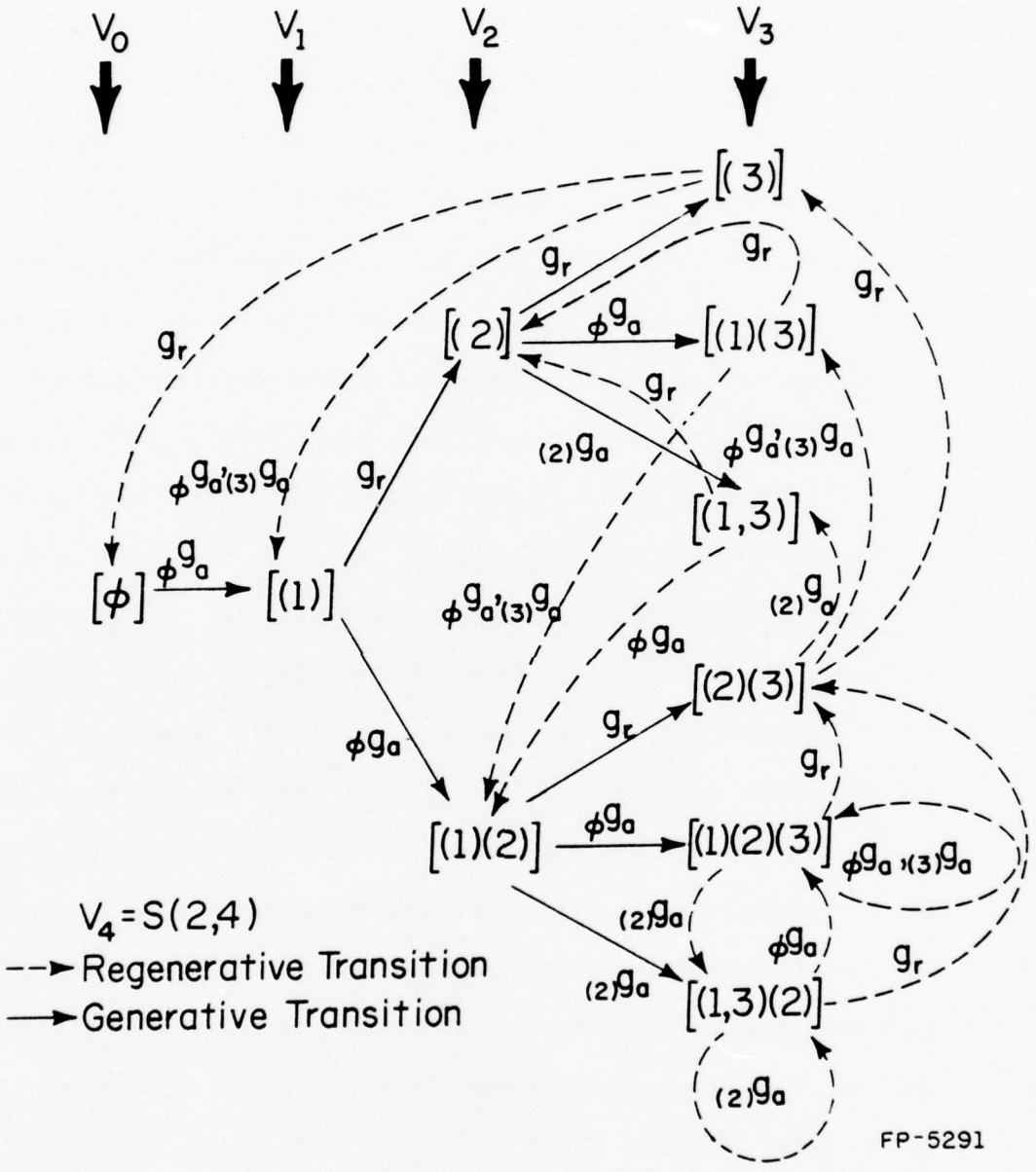


Figure 3.2.1 Generation of System State Graph, $G_s(2, 4)$, for (a, c) = (2, 4)

For convenience, however, we will denote the probability of transition from system state $\sigma(t)$ to its successor acceptance state, ${}_{\lambda}g_a(\sigma)$, by ${}_{\lambda}p_a(\sigma)$. Similarly, we will denote the probability of transition from system state $\sigma(t)$ to its successor rejection state, $g_r(\sigma)$, by $p_r(\sigma)$. Such transition probabilities imply that the state of the system at any time instant, t, is stochastically determined by its state at the preceding time instant. The transition probabilities from a particular system state, $\sigma(t)$, to all states in the system, $S(a, c)$, must add up to 1. Since the transitions from $\sigma(t)$ are to its successor rejection state and acceptance states,

$$\left(\sum_{\substack{\text{idle line states,} \\ \lambda \text{ in } \sigma(t)}} {}_{\lambda}p_a(\sigma) \right) + p_r(\sigma) = 1.$$

Theorem 3.2.7 The probability of transition from the system state $\sigma(t)$ to its successor rejection system state, $g_r(\sigma) = \sigma_n(t + 1)$ is

$$p_r(\sigma) = \frac{m\,k + j}{N}$$

where k = total number of busy line states in $\sigma(t)$;

j = total number of elements in idle line states in $\sigma(t)$;

m = total number of memory modules on a line; and

N = total number of memory modules in the system.

Proof A request is rejected by the system if the request addresses any of the busy lines or busy modules on idle lines at t. The probability of a request addressing a busy line at t is the total number of busy lines at t/total lines in the system. The total number of busy lines at t = total number of busy line states in $\sigma(t)$ = k. Hence the probability of a request addressing a busy line at t = k/ℓ. The

probability of a request addressing a busy module on an idle line at t = (the total number of busy modules on idle lines at t)/(total number of modules in the system). The total number of busy modules on idle lines at t = total number of nonnull module states in idle line states in $\sigma(t)$ = $\sum_{\lambda(t) \in \sigma(t) \mid \lambda(t) = \text{idle line state}} (|\lambda(t)|) = j$.

Hence the probability of a request addressing a busy module on an idle line at t = j/N. Therefore, $p_r(\sigma) = \frac{k}{\ell} + \frac{j}{N}$. Since $N = \ell m$, $p_r(\sigma) = (mk + j)/N$. □

For example, consider the computation of $p_r([(1)(2)])$ and $p_r([(2)(3)])$ for (a, c) = (2, 4). For [(1)(2)], k = 1 and j = 1, hence $p_r([(1)(2)]) = (m + 1)/N$. For [(2)(3)], k = 0 and j = 2, hence $p_r([(2)(3)]) = 2/N$.

Evaluation of the probability of transition from the system state $\sigma(t) \in S(a, c)$ to its successor acceptance state is not as straightforward as that for the successor rejection state because there are different possible successor acceptance states. Moreover, for regenerative transitions, we recall that references to more than one idle line state in $\sigma(t) \in V_{c-1}$ may result in transitions to the same acceptance state. For the (a, c) = (2, 4) example, [(1)(2)(3)] makes a transitions to [(1)(2)(3)] if the request references the idle line state, Ø. Similarly, [(1)(2)(3)] can also make a transition to [(1)(2)(3)] if the request references the idle line state, (3).

In general, a generative system state $\sigma(t)$ makes a generative transition to $\sigma_\lambda(t + 1) = {}_\lambda g_a(\sigma) = [(1)] \cup g_r(\sigma)$, if and only if the request references the idle line state, $\lambda = \emptyset \in \sigma(t)$. However, a regenerative

system state, $\sigma(t)$, makes a regenerative transition to

$$\sigma_\lambda(t+1) = {}_\lambda g_a(\sigma) = [(1)] \cup g_r(\sigma)$$

if and only if the request references the idle line state $\lambda = \emptyset$ or $\lambda = (c-1)$, both of which may be in $\sigma(t)$.

Theorem 3.2.8 The probability of transition from the system state $\sigma(t)$ to its successor acceptance state of the form $\sigma_\lambda(t+1) = {}_\lambda g_a(\sigma)$ is

$$ {}_\lambda p_a(\sigma) = \frac{N - |\sigma|\, m}{N} \text{ , if } \lambda = \emptyset \text{ ,}$$

$$= \frac{m - |\lambda|}{N} \text{ , otherwise.}$$

Proof The probability of transition from system state $\sigma(t)$ to $\sigma_\lambda(t+1) = [(1)] \cup g_r(\sigma)$, is the probability of a request referencing a line with a null line state at t, which is equal to (the total number of lines with null line states at t)/ℓ. The total number of lines with null line states at t = the total number of lines with no busy modules at t = ℓ - (the total number of lines with busy modules at t) = $\ell - |\sigma|$. Hence the probability of referencing a line with a null line state at $t = (\ell - |\sigma|)/\ell = (N - |\sigma|\, m)/N$, since $N = \ell m$. Therefore, ${}_\lambda p_a(\sigma) = (N - |\sigma|\, m)/N$, if transition is only due to acceptance on a line with no busy modules.

The probability of transition from a system state $\sigma(t)$ to ${}_\lambda g_a(\sigma)$, where $\lambda \neq \emptyset$ is the probability of a request referencing an idle module on the idle line represented by $\lambda(t) \in \sigma(t)$. The probability of a request referencing an idle module on that line given that it addresses that line at t = (the total number of idle modules on that line at t)/m. The total number of idle modules on that line at t = m - (the total number

of busy modules on that line at t) = $m - |\lambda| \cdot {}_{\lambda}P_a(\sigma)$ = (the probability of a request referencing the line at t) x (the probability of the request referencing an idle module on the line at t). Therefore,

$${}_{\lambda}P_a(\sigma) = \frac{1}{\ell} \cdot \frac{m - |\lambda|}{m} = \frac{m - |\lambda|}{N} .$$

□

Notice however, that for regenerative transitions, transition from $\sigma(t)$ to $[(1)] \cup g_r(\sigma)$ may be due to acceptance on a line with no busy modules or on a line with one busy module in state (c - 1). Hence both transition probabilities, ${}_{\emptyset}P_a(\sigma)$ and ${}_{(c-1)}P_a(\sigma)$ go to the same state, $[(1)] \cup g_r(\sigma)$. Therefore the transition probability from $\sigma(t)$ to state $[(1)] \cup g_r(\sigma)$ is ${}_{\emptyset}P_a(\sigma) + {}_{(c-1)}P_a(\sigma) = (N - |\sigma| m)/N + (m-1)/N = (N - (|\sigma| - 1)m - 1)/N$.

For the (a, c) = (2, 4) example, the probability of transition from [(1) (2)] to [(1)(2)(3)] is ${}_{\emptyset}P_a([(1)(2)]) = (N - 2m)/N$. Since transition from σ = [(1)(2)(3)] to [(1)(2)(3)] may be due to ${}_{\emptyset}P_a(\sigma)$ or ${}_{(3)}P_a(\sigma)$, the transition probability from [(1)(2)(3)] to [(1)(2)(3)] is ${}_{\emptyset}P_a(\sigma) + {}_{(3)}P_a(\sigma) = (N - 2m - 1)/N$. The probabilities of transition from [(1)(2)(3)] to [(1,3) (2)] and from [(1, 3)(2)] to [(1,3) (2)] are each $(m-1)/N$. Recognize that in both cases, the idle line state is (2). Figure 3.2.2 shows the system state graph, $G_s(2,4)$ in which each arc is labeled with the corresponding probability of transition.

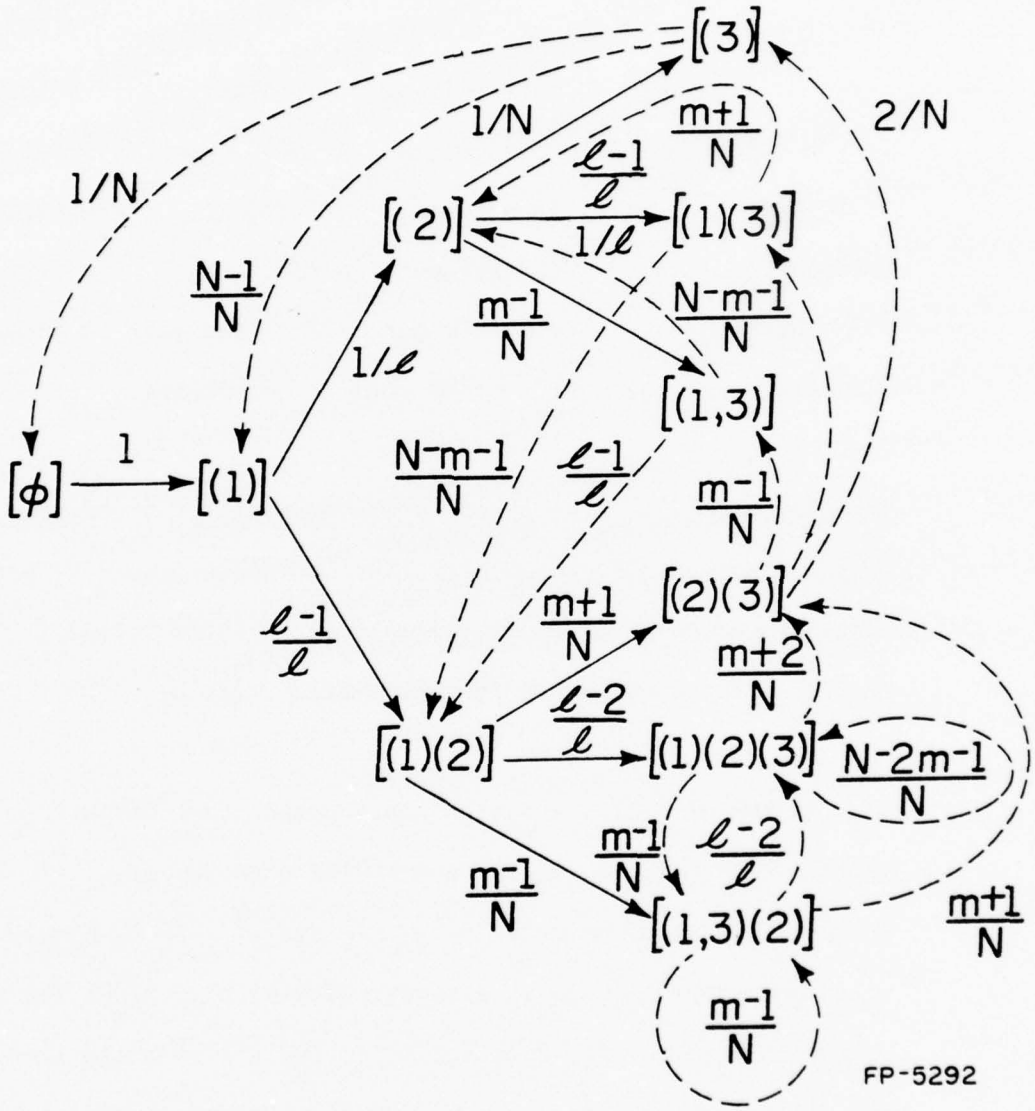


Figure 3.2.2 System State Graph, $G_s(2, 4)$, with Transition Probabilities

3.3 State Reduction and Line Decomposition

A brief investigation of system state graphs reveals that the number of states in a system state graph, for sufficiently large values of ℓ and m, increases dramatically as c increases relative to a. For example, if the number of system states in S(a, c) is denoted by $|S(a, c)|$, then $|S(2, 2)| = 2$, $|S(2, 4)| = 10$ and $|S(2, 6)| = 57$. However, obtaining a formula for $|S(a, c)|$ is very complicated.

The main objective of the performance analysis is to obtain the performance of the system under various parameters.

Definition 3.3.1 The steady state probability of acceptance, $P_A(a, c, p)$, is the steady state probability that a request issued by a parallel-pipelined processor of order (s, p) will be accepted by the (ℓ, m) memory configuration with module characteristics, (a, c).

As the size of the system state graph grows for $a > 1$, the complexity of computing $P_A(a, c, p)$ for $p = 1$ also grows. For $p = 1$, $P_A(a, c, p)$ is the probability of being in a certain set of states, namely, the set of acceptance states which is a subset of S(a, c). Since the objective is to obtain $P_A(a, c, 1)$, a reduction of the system state graph is possible by collapsing the states which literally appear identical to a request attempting to access the system in those states.

In this section an attempt is made to discover and eliminate unnecessary states in the state graph. It is also shown that since all lines in the memory system are identical and independent, a single line

model, instead of the total system model used thus far, can be developed to simplify analysis of the system significantly.

Before the system state reduction is discussed, certain relationships between line states in the set of line states $\Lambda(a, c)$ will be investigated.

Recall that a busy line may have more than one busy module on it. Consider the case for $(a, c) = (2, 4)$ in which a request addresses a busy line represented by the busy line state $\lambda = (1, 3)$. The request will be rejected causing a transition from $(1, 3)$ to $f_n(1, 3) = (2)$. The same next state would result if the addressed line was represented by (1). Therefore, there may be some cases, as demonstrated above, in which the next line states of two distinct busy line states are identical.

Let $f_n^j(\lambda)$ represent the nonacceptance state of λ arrived at by successive application of the line nonacceptance function j times, for an integer $j \geq 0$. That is, $f_n^j(\lambda) = f_n(f_n^{j-1}(\lambda))$, where $f_n^1(\lambda) = f_n(\lambda)$ and $f_n^0(\lambda) = \lambda$.

Definition 3.3.2 Two line states, λ_1 and λ_2, in $\Lambda(a, c)$ are equivalent, written $\lambda_1 \sim \lambda_2$, if they are identical or if they have identical smallest nonnull element $r_{min} < a$, such that $f_n^j(\lambda_1) = f_n^j(\lambda_2)$, for $j = (a - r_{min})$. □

This definition includes idle line states. However, an idle line state $\lambda \in \Lambda(a, c)$ is only equivalent to itself, i.e., $\lambda \sim \lambda$. Although each line state is equivalent to itself, it is easy to test for equivalence

with other line states by applying the definition. However, only busy line states need be considered to find equivalence among distinct states. Moreover, for distinct busy line states, only pairs with identical smallest element need be compared.

Theorem 3.3.1 If $\lambda_1 \sim \lambda_2$ then $f_n^i(\lambda_1) \sim f_n^i(\lambda_2)$, for $i \geq 0$.

Proof: If $\lambda_1 = \lambda_2$, the proof is trivial since $f_n^i(\lambda_1) = f_n^i(\lambda_2)$, $\forall\ i \geq 0$. However, if $\lambda_1 \neq \lambda_2$, for $j = (a - r_{min}) > 0$, where r_{min} is the identical smallest element in λ_1 and λ_2, $f_n^j(\lambda_1) = f_n^j(\lambda_2)$. We need to show that if $\lambda_1 \sim \lambda_2 \Rightarrow f_n(\lambda_1) \sim f_n(\lambda_2)$. Figure 3.3.1 illustrates the problem. Since $\lambda_1 \neq \lambda_2$ and $\lambda_1 \sim \lambda_2$, $a > r_{min}$. From theorem 3.2.1, $f_n(\lambda_1) = (x \mid x - 1 \in \lambda_1(t)$ and $x < c)$. Hence the identical smallest element in $f_n(\lambda_1)$ and $f_n(\lambda_2)$ is $r_{min} + 1 = r'_{min}$. Let $j' = (a - r'_{min}) \geq 0$. Note that $j' = j - 1$. Since $f_n^j(\lambda_1) = f_n^j(\lambda_2)$, $f_n^{j-1}(f_n(\lambda_1)) = f_n^{j'}(f_n(\lambda_1)) = f_n^{j-1}(f_n(\lambda_2)) = f_n^{j'}(f_n(\lambda_2))$, for $j' \geq 0$. Hence $f_n(\lambda_1) \sim f_n(\lambda_2)$. Therefore by i applications, $\lambda_1 \sim \lambda_2 \Rightarrow f_n^i(\lambda_1) \sim f_n^i(\lambda_2)$, $\forall\ i \geq 0$. □

This theorem states that if two line states are equivalent, their successor states are also equivalent and possibly identical. For example, consider the two distinct line states λ_1=(1,4) and λ_2=(1,5) in $\Lambda(3,6)$. $j = a - r_{min} = 3 - 1 = 2$ and $f_n^2(1, 4) = (3) = f_n^2(1, 5)$. Hence $(1, 4) \sim (1, 5)$. Therefore, $f_n^i(1, 4) \sim f_n^i(1, 5)$, for $i \geq 0$. Hence $(2, 5) \sim (2)$, $(3) \sim (3)$, $(4) \sim (4)$, and so forth. In common terms, two busy line states are equivalent if they differ only in elements large enough to be incremented beyond c - 1 by the time the line is idle.

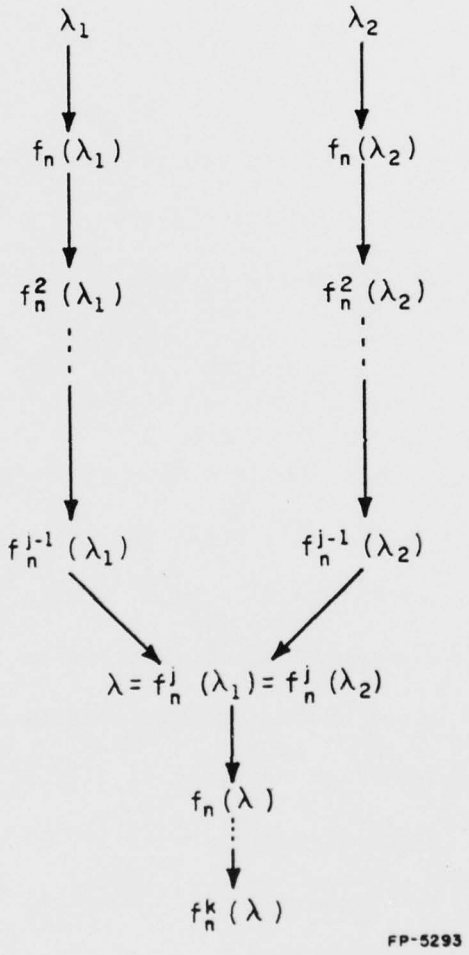


Figure 3.3.1 Illustration of Theorem 3.3.1

The equivalence relation defined above on $\Lambda(a, c)$ partitions $\Lambda(a, c)$ into a set of equivalence classes, $\xi_\lambda(a, c)$, called line equivalence classes. All states in each class are equivalent to each other and no state in one class is equivalent to any state in any other class. For example, the line equivalence classes, $\xi_\lambda(2, 4)$ for $(a, c) = (2, 4)$ are $\{\emptyset\}$, $\{(1), (1, 3)\}$, $\{(2)\}$ and $\{(3)\}$.

Definition 3.3.3 Two system states, σ_1 and σ_2 in $S(a, c)$, are equivalent, written $\sigma_1 \sim \sigma_2$, if for every $\lambda_i \in \sigma_1$, $\exists$ a unique $\lambda_k \in \sigma_2$ such that $\lambda_i \sim \lambda_k$ and for every $\lambda_k \in \sigma_2$, $\exists$ a unique $\lambda_i \in \sigma_1$ such that $\lambda_k \sim \lambda_i$. □

Since all the elements of a system state are distinct, $|\sigma_1| = |\sigma_2|$. For example, in $S(3, 6)$, $[(1, 4)(2, 5)(3)] \sim [(1)(2)(3)]$, since $(1, 4) \sim (1)$, $(2, 5) \sim (2)$ and $(3) \sim (3)$. As with line state equivalence, system state equivalence is an equivalence relation on $S(a, c)$ and hence partitions $S(a, c)$ into a set of equivalence classes, $\xi_\sigma(a, c)$, called system equivalence classes. For the example of $S(2, 4)$, the system equivalence classes are $\{[\emptyset]\}$, $\{[(1)], [(1, 3)]\}$, $\{[(2)]\}$, $\{[(1)(2)], [(1, 3)(2)]\}$, $\{[(3)]\}$, $\{[(1)(3)]\}$, $\{[(2)(3)]\}$ and $\{[(1)(2)(3)]\}$.

In order to determine $\xi_\sigma(a, c)$, it is only necessary to consider pairs of system states whose cardinalities are identical and have identical sets of smallest elements in their line states. The reduced set of system states, $S'(a, c)$, represents the system equivalence classes for the system state graph, $G_s(a, c)$. Any state in an equivalence class can be

selected to represent the set of states in the equivalence class. However, for consistency, the state with the least number of module states in each equivalence class will be selected as the system state representing that equivalence class. In the $\xi_\sigma(2, 4)$ example, [(1)] would represent {[(1)], [(1, 3)]} and [(1)(2)] would represent {[(1)(2)], [(1, 3)(2)]}.

With the reduced set of states, S'(a, c), the reduced state graph can be obtained by merging each state not in S'(a, c) with its equivalent state in S'(a, c). By so doing, the edges inbound to such a state not in S'(a, c) are transferred to the state's equivalent in S'(a, c). The graph thus obtained is the reduced state graph, $G'_s(a, c)$. Notice that if any two states, σ_j, $\sigma_k \in S(a, c)$, are equivalent such that $\sigma_j \in S'(a, c)$ but $\sigma_k \notin S'(a, c)$, then the transfer of edges to σ_j, which were inbound to σ_k, implies that the new transition probability, p_{ij}, from any system state $\sigma_i \in S'(a, c)$ to σ_j in $G'_s(a, c)$, is the old transition probability, p_{ik} in $G_s(a, c)$.

The algorithm for obtaining $G'_s(a, c)$ from $G_s(a, c)$ is summarized below.

Step 1. Form $\xi_\sigma(a, c)$: Partition S(a, c) into system equivalence classes.

Step 2. Form S'(a, c): In each equivalence class, select the state with the least number of module states.

Step 3. Draw all outgoing edges from each state in S'(a, c) as in $G_s(a, c)$ except outgoing edges that terminate at states not in S'(a, c).

Step 4. Redirect all incoming edges to states not in S'(a, c) to their respective equivalent states in S'(a, c).

Applying the algorithm to the example $G_s(2, 4)$ of figure 3.2.2, which is repeated in figure 3.3.2(a), produces the reduced system state graph, $G'_s(2, 4)$, shown in figure 3.3.2(b). From the reduced state graph, $G'_s(2, 4)$, the steady state probability of acceptance $P_A(a, c, p)$ for $p = 1$ and any (ℓ, m) memory configuration can be obtained for the module characteristics $(a, c) = (2, 4)$.

$$P_A(2, 4, 1) = \frac{N^2}{N^2 + Nm + 2N - m + 1}$$

where $N = \ell m$. $P_A(2, 4, 1)$ is the sum of the probabilities of being in the acceptance states [(1)], [(1)(2)], [(1)(3)] and [(1)(2)(3)] in $G'_s(2, 4)$. The detailed computation of $P_A(2, 4, 1)$ follows standard Markov analysis techniques, but is fairly complicated as shown in Appendix A.

In general, the computation of $P_A(a, c, 1)$ can be much simplified if it is observed that the ℓ lines of the memory system are identical and independent. Since it was assumed that requests are uniformly distributed among the ℓ lines, a line decomposed model would suffice to obtain the steady state probability of acceptance, $P_A(a, c, p)$, for $p = 1$. A line decomposed model is the Markov model of one line.

The line decomposed model is now investigated with an aim toward obtaining a simpler model for obtaining $P_A(a, c, 1)$. The set of line states, $\Lambda(a, c)$, are the states of the line decomposed model for module characteristics (a, c).

A <u>line state graph</u>, $G_\ell(a, c)$, which is similar to a system state graph, can be used to display the next line state function. It consists of one node for each line state in $\Lambda(a, c)$ and one edge leaving each

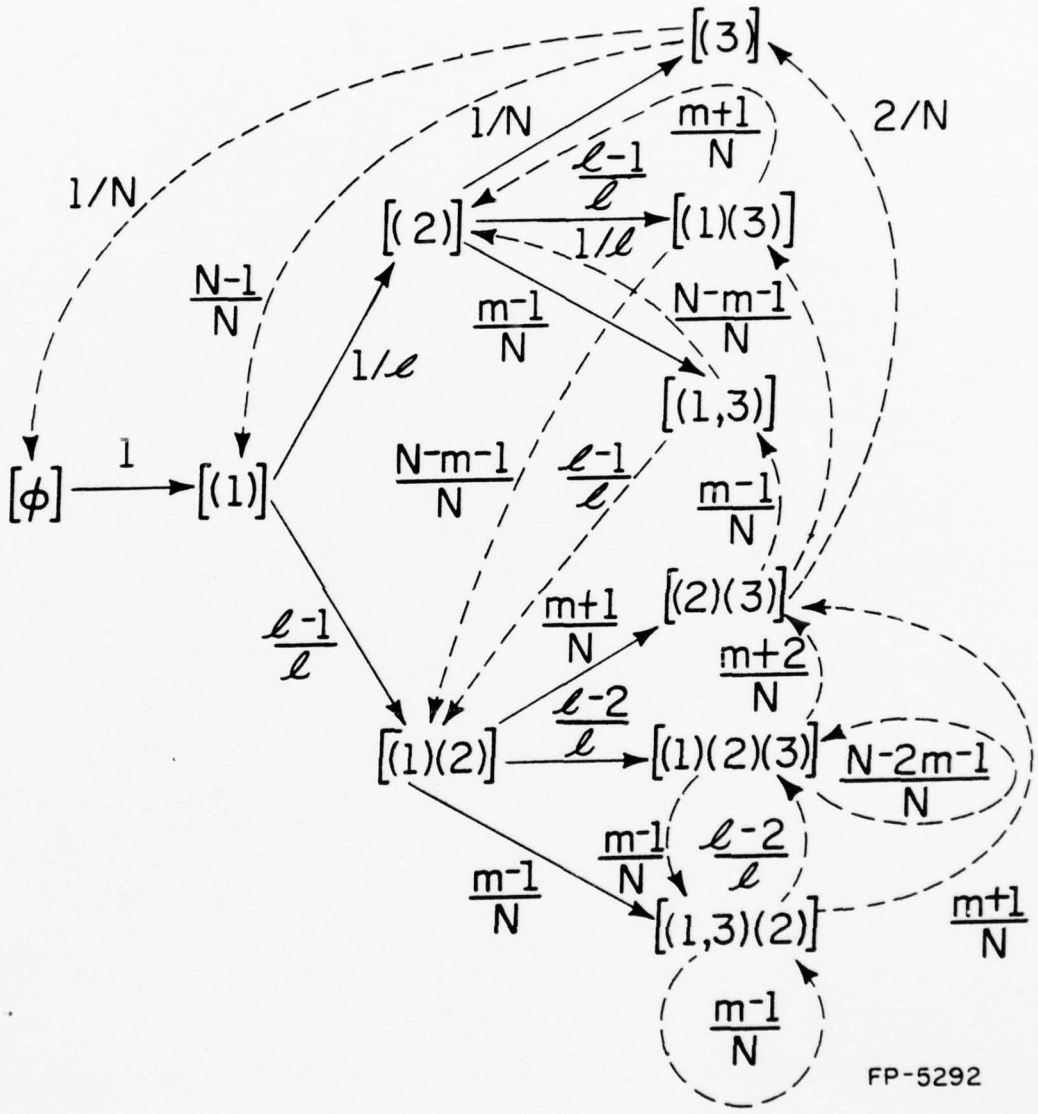


Figure 3.3.2(a) System State Graph, $G_s(2, 4)$

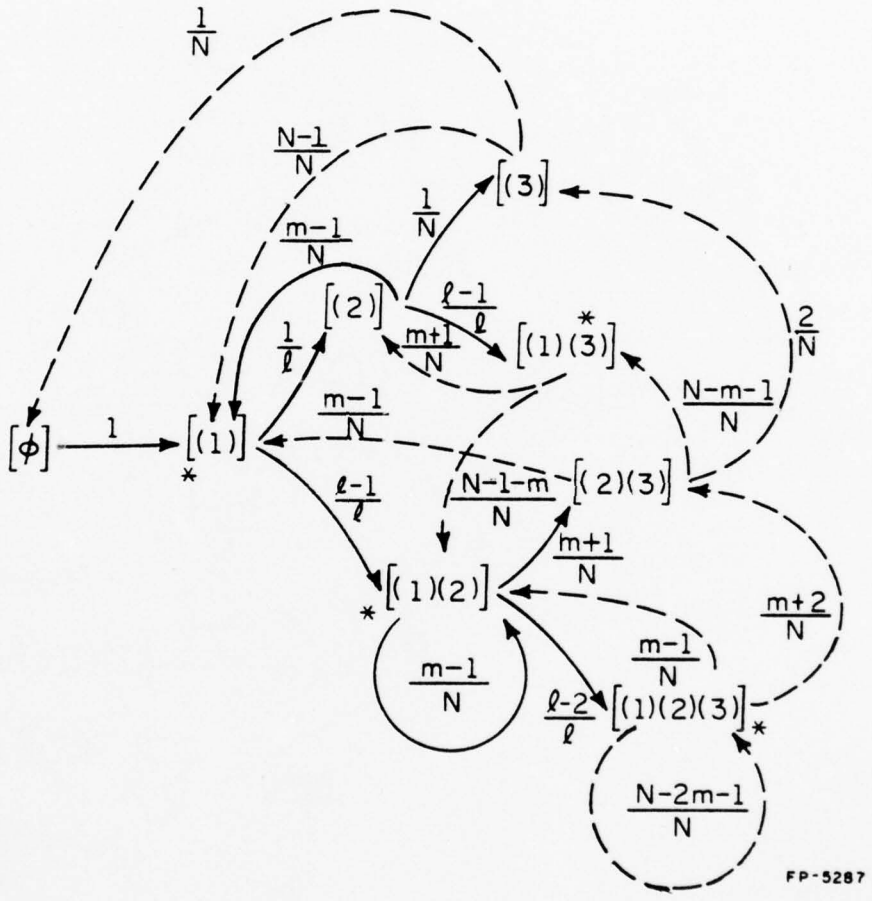


Figure 3.3.2(b) Reduced System State Graph, $G_s'(2, 4)$

node for each possible acceptance or nonacceptance transition. A simple technique is given below to generate the line state graph for module characteristics (a, c).

Let $W_o = \{\emptyset\}$ and W_n be the set of new line states generated by a one-step transition from the line states in W_{n-1} for $1 \leq n \leq c-1$. Furthermore, W_c is the set of line states generated by a one-step transition from line states in W_{c-1}. The set of nonacceptance states generated by states in W_{n-1} is R_n, where $R_n = \{\lambda | \lambda = \lambda_n(\lambda_i), \lambda_i \in W_{n-1}\}$. Similarly, the set of acceptance states generated by states in W_{n-1} is A_n, where $A_n = \{\lambda \mid \lambda = f_a(\lambda_i), \lambda_i \in W_{n-1}$ and λ_i = idle line state$\}$. $R_1 = \{\emptyset\}$ and $A_1 = \{(1)\}$. $W_n = A_n$ for $n = 1$, and $W_n = A_n \cup R_n$, for $1 < n \leq c - 1$. Notice that R_1 is empty since the state $\emptyset \in W_o$ is not new.

Theorem 3.3.2 W_{c-1} is the set of all regenerative line states for given (a, c).

□

This theorem is similar to Theorem 3.2.5 whose proof is given. Hence, $W_0, W_1, \ldots, W_{c-1}$ generate all the line states while simultaneously producing the generative transitions of the line state graph. Thus $\Lambda(a, c) = \bigcup_{i=0}^{c-1} W_i$. Recall that, for $1 \leq i \leq c - 1$, the largest element in any line state in W_i is i, hence transitions from line states in W_{i-1} to W_i are generative. In particular, transition from the line state, $\emptyset$ to $\emptyset$ is regenerative. Transitions from line states in W_{c-1} to W_c are regenerative. Hence, the generation of A_c and R_c produces all the regenerative transitions and thereby completes the line state diagram.

Theorem 3.3.3 $W_c \subseteq \Lambda(a, c)$ □

The proof of this theorem is trivial since transitions from W_{c-1} are regenerative.

As an example, consider the formation of $G_\ell(2, 4)$. Figure 3.3.3 shows the line state graph, $G_\ell(2, 4)$. Nodes marked "*" are acceptance line states. $W_o = \{\emptyset\}$, then $R_1 = \emptyset$ and $A_1 = \{(1) = f_a(\emptyset)\}$. $W_1 = \{(1)\}$. $R_2 = \{(2) = f_n(1)\}$ and $A_2 = \emptyset$, hence $W_2 = A_2 \cup R_2 = \{(2)\}$. $R_3 = \{(3) = f_n(2)\}$ and $A_3 = \{(1, 3) = f_a(2)\}$, hence $W_3 = \{(3), (1, 3)\}$. This completes the generation phase. The regeneration phase is given by $R_4 = \{\emptyset = f_n(3), (2) = f_n(1, 3)\}$ and $A_4 = \{(1) = f_a(3)\}$. The transition probabilities are obtained with the aid of the following theorem.

Theorem 3.3.4 The probability of transition from an idle line state $\lambda(t)$ to its successor acceptance line state is

$$p_a(\lambda) = \frac{m - |\lambda|}{N}$$

where $|\lambda|$ is the cardinality of the line state $\lambda(t)$.

Proof: Since $\lambda(t)$ is an idle line state, a request which addresses the line corresponding to $\lambda(t)$ will be accepted if the request addresses an idle module on the line in question. The number of idle modules on the line represented by $\lambda(t)$ is $m - |\lambda|$, where $|\lambda|$ is the number of busy modules on the line. Given a request to the line, the conditional probability of requesting any one of the idle modules on the line = $(m - |\lambda|)/m$. Since the probability of requesting the line is $1/\ell$, $p_a(\lambda) = \frac{1}{\ell} \cdot \frac{(m - |\lambda|)}{m} = \frac{m - |\lambda|}{N}$. □

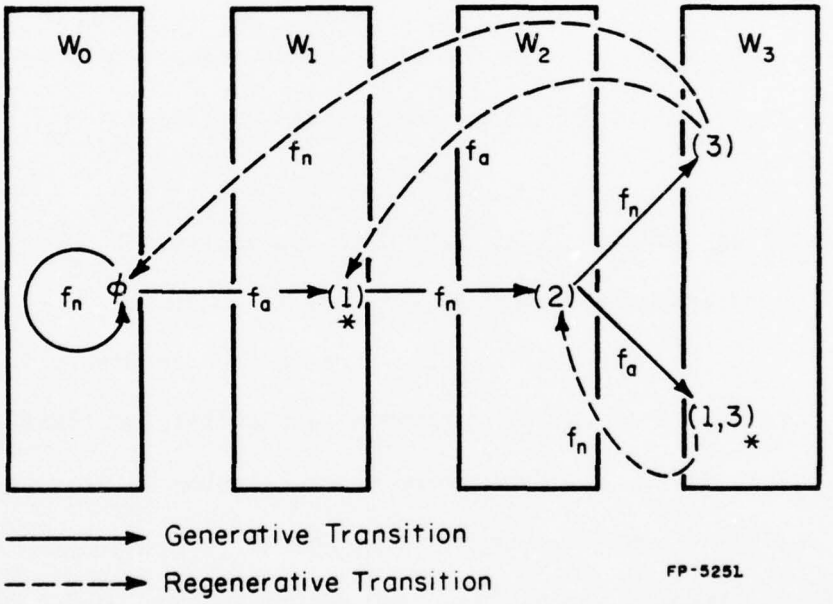


Figure 3.3.3 Generation of Line State Graph, $G_\ell(2, 4)$ for (a, c) = (2, 4)

Corollary 3.3.4.1 The probability of transition from an idle line state, $\lambda(t)$, to its successor nonacceptance state is

$$p_n(\lambda) = 1 - p_a(\lambda).$$ □

This is obvious since $p_n(\lambda) + p_a(\lambda) = 1$. Note that nonacceptance implies either rejection of a request to the line or arrival of a request to some other line.

Theorem 3.3.5 The probability of transition from a busy line state, $\lambda(t)$, to its successor nonacceptance state is $p_n(\lambda) = 1$. □

Since $\lambda(t)$ is a busy line state, there is no successor acceptance state hence the result follows.

The example of $G_\ell(2, 4)$ is redrawn in figure 3.3.4 with the arcs labeled with the corresponding probability of transition.

Recall that the equivalence relation on $\Lambda(a,c)$ partitions $\Lambda(a,c)$ into disjoint equivalence classes. Hence in general, $\Lambda(a, c)$ can be reduced if there exists at least two states in $\Lambda(a, c)$ that are equivalent to each other. The reduced line state graph, $G'_\ell(a, c)$, can be obtained from $G_\ell(a, c)$ following an algorithm very similar to that used in obtaining $G'_s(a, c)$ from $G_s(a, c)$. For the example of $G_\ell(2, 4)$, figure 3.3.5 shows the reduced line state graph, $G'_\ell(2, 4)$, where equivalent line states (1, 3) and (1) are merged.

From the reduced line state graph, $G'_\ell(2, 4)$, the steady state probability of being in each of the four states can be calculated. However,

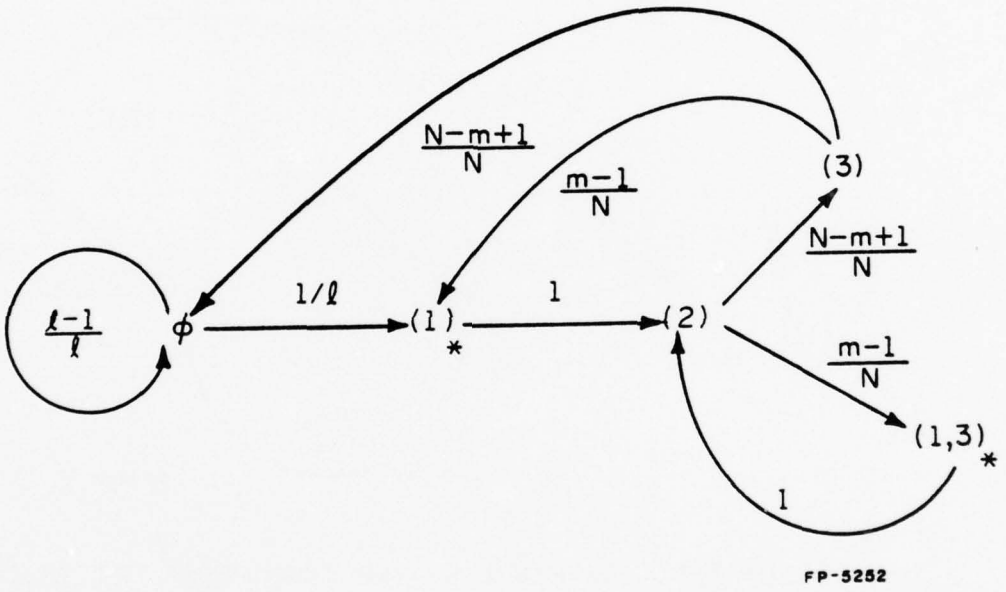


Figure 3.3.4 Line State Graph, $G_\ell(2, 4)$, with Transition Probabilities

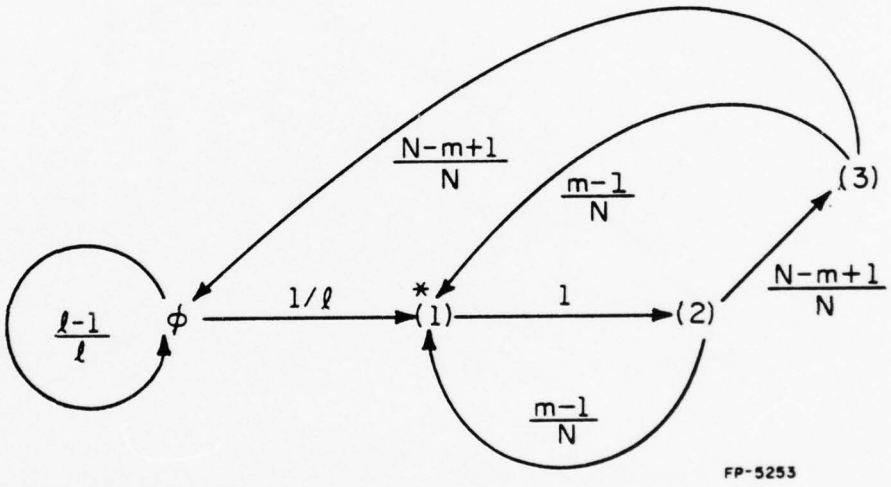


Figure 3.3.5 Reduced Line State Graph, $G'_{\ell}(2, 4)$

the crux of the analysis is in calculating the steady state probability, $P_{A\ell}(a, c, p)$, of being in an acceptance line state in the line decomposed system for p = 1. In $G'_{\ell}(2, 4)$, the only acceptance line state is (1). Let P_{λ} denote the probability of being in the line state, λ. Hence, for p = 1,

$$P_{A\ell}(a, c, p) = \sum_{\text{acceptance line states, } \lambda \mid \lambda \in G'_{\ell}(a,c)} P_{\lambda}$$

Then from $G'_{\ell}(2, 4)$,

$$P_{\emptyset} = \frac{\ell - 1}{\ell} P_{\emptyset} + \frac{N - m + 1}{N} P_{(3)} \quad \ldots \ldots \ldots \ldots \ldots \ldots \ldots \quad 1.$$

$$P_{(1)} = \frac{1}{\ell} P_{\emptyset} + \frac{m - 1}{N} [P_{(3)} + P_{(2)}] \quad \ldots \ldots \ldots \ldots \ldots \ldots \quad 2.$$

$$P_{(2)} = P_{(1)} \quad \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \quad 3.$$

$$P_{(3)} = \frac{N - m + 1}{N} P_{(2)} = \frac{N - m + 1}{N} P_{(1)} \quad \ldots \ldots \ldots \ldots \ldots \quad 4.$$

$$P_{\emptyset} + P_{(1)} + P_{(2)} + P_{(3)} = 1 \quad \ldots \ldots \ldots \ldots \ldots \ldots \quad 5.$$

From equations 1 and 4, $P_{\emptyset} = \frac{\ell(N - m + 1)}{N} P_{(3)} = \frac{\ell(N - m + 1)^2}{N^2} P_{(1)}$.

Substituting for $P_{\emptyset}$, $P_{(2)}$, and $P_{(3)}$ in equation 5,

$$\left[\frac{\ell(N - m + 1)^2}{N^2} + 1 + 1 + \frac{N - m + 1}{N}\right] P_{(1)} = 1$$

Rearranging,

$$\frac{\ell(N - m + 1)^2 + 2N^2 + (N - m + 1)N}{N^2} P_{(1)} = 1$$

Hence

$$P_{(1)} = \frac{N^2}{(N - m + 1)\,[\ell(N - m + 1) + N] + 2N^2}$$

Since $\ell m = N$,

$$P_{(1)} = \frac{N^2}{(N - m + 1)(N + 1)\ell + 2N^2} = \frac{N^2}{(N^2 + Nm + 2N - m + 1)\ell}$$

Notice that in the reduced line state graph, $G'_{\ell}(2, 4)$, $\lambda = (1)$ is the only acceptance state. Hence, in this example, $P_{A\ell}(2, 4, 1) = P_{(1)}$. In general, $P_{A\ell}(a, c, p)$ is the probability of acceptance of a request on the line being modeled.

Theorem 3.3.6 For $p = 1$, the steady state probability of acceptance of a request in the L-M memory organization is

$$P_A(a, c, 1) = \ell P_{A\ell}(a,c,1) \qquad \square$$

This is obvious since all the ℓ lines of the L-M memory organization are identical and independent.

In the example of $(a, c) = (2, 4)$, $P_{A\ell}(a,c,1) = P_{(1)}$. Hence,

$$P_A(2, 4, 1) = \ell \frac{N^2}{(N^2 + Nm + 2N - m + 1)\ell} = \frac{N^2}{N^2 + Nm + 2N - m + 1},$$

which is identical to the results obtained from the reduced system state graph, $G'_s(2, 4)$, but less tediously obtained.

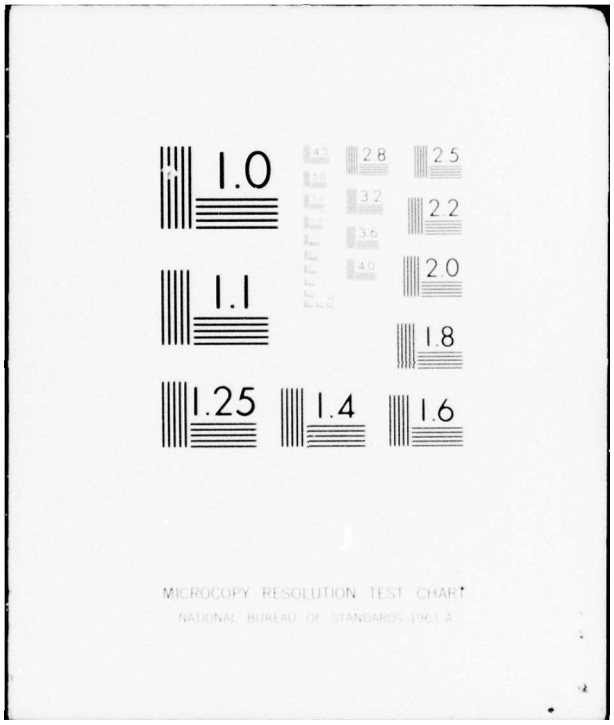
Hence in conclusion, the line decomposition of the system allows a vast simplification of the performance analysis problem. Line decomposition will be adopted in the rest of this analysis of the L-M memory organization.

3.4 Line State Space

Although the number of line states is much less than the number of system states for any module characteristics (a, c), a brief investigation of the line state graph, $G_{\ell}(a, c)$, also reveals that the number of line states in a line state graph increases dramatically as c increases

AD-A042 646 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
MEMORY ORGANIZATIONS AND THEIR EFFECTIVENESS FOR MULTIPROCESSIN--ETC(U)
MAY 77 F A BRIGGS DAAB07-72-C-0259
UNCLASSIFIED R-768 NL

2 OF 3
AD
A042646


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

relative to a. The complexity of computing $P_A(a, c, 1)$ increases with the size of the line state graph.

In this section, we will characterize the number of line states in a line state graph, $G_\ell(a, c)$. It is also shown that the number of line states in $G_\ell(a, c)$ depends on the maximum number of busy modules on a line.

Theorem 3.4.1 The maximum number of busy modules on a line for module characteristics (a, c) is $\left\lceil \frac{c - 1}{a} \right\rceil$.

Proof: As usual, let the element of a line state be listed in ascending order. It is easy to show that the maximum number of busy modules on a line occurs when the line is in the acceptance state $\lambda = (1, a + 1, 2a + 1, \ldots, ka + 1)$, where k is the greatest integer such that $ka + 1 \leq c - 1$. Notice that in this state adjacent elements have a difference of exactly a. It should be pointed out that this acceptance state may not be the only line state with the maximum number of busy modules. The maximum number of elements, and hence the maximum number of busy modules, in the acceptance state λ is $k + 1$. We know that for an integer, n, and a real number, x, $\lceil x \rceil = n$, if $x \leq n < x + 1$. In order to show that this is true for $x = \frac{c - 1}{a}$ and $n = k + 1$, we know that $ka + 1 \leq c - 1 \Rightarrow ka < ka + 1 \leq c - 1$, for integers k and a such that $k \geq 0$ and $a \geq 1$. Hence $ka < c - 1 \Rightarrow k < \frac{c - 1}{a} \Rightarrow k + 1 < \frac{c - 1}{a} + 1$. Since k is the greatest integer such that $ka + 1 \leq c - 1$ and $a \geq 1$, then $(k + 1)a + 1 > c - 1$. That is, $k + 1 > \frac{c - 1}{a} - \frac{1}{a}$. Let q and r be integers such that $0 \leq r \leq a - 1$ and $\frac{c - 1}{a} - \frac{1}{a} = q + \frac{r}{a}$. Hence, $\frac{c - 1}{a} = q + (\frac{r}{a} + \frac{1}{a})$ and $0 < \frac{r}{a} + \frac{1}{a} \leq 1$.

Since k is the greatest integer such that $k + 1 > q + \frac{r}{a}$ and $0 < \frac{r}{a} + \frac{1}{a} \leq 1$, it implies that $k + 1 \geq q + \frac{r}{a} + \frac{1}{a}$. Hence $k + 1 \geq \frac{c - 1}{a}$. Combining the above results, $\frac{c - 1}{a} \leq k + 1 < \frac{c - 1}{a} + 1$. Therefore $k + 1 = \left\lceil \frac{c - 1}{a} \right\rceil$. □

Hence $|\lambda| \leq \left\lceil \frac{c - 1}{a} \right\rceil$ for $\lambda \in \Lambda(a, c)$.

Theorem 3.4.2 The maximum number of busy modules on an idle line for module characteristics (a, c) is $\left\lfloor \frac{c - 1}{a} \right\rfloor$.

Proof: It is trivial to show that the theorem holds for $a = c = 1$. Assume that $c \geq a \geq 1$. The maximum number of busy modules on an idle line occurs when the idle line state is $\lambda = (a, 2a, 3a, \ldots, ka)$, where k is the greatest integer such that $ka \leq c - 1$. Such a k exists since $a \geq 1$. Then $(k + 1)a > c - 1$. Hence $k = \left\lfloor \frac{c - 1}{a} \right\rfloor$. □

Hence $|\lambda| \leq \left\lfloor \frac{c - 1}{a} \right\rfloor$, for an idle line state $\lambda \in \Lambda(a, c)$. Recall from the last section that the transition probability of an idle line state $\lambda(t)$ to its successor acceptance state is $p_a(\lambda) = (m - |\lambda|)/N$. Hence $(m - \left\lfloor \frac{c - 1}{a} \right\rfloor)/N \leq p_a(\lambda) \leq 1$, for $G_\ell(a, c)$. It will be shown in the next section that it is this number, $\left\lfloor \frac{c - 1}{a} \right\rfloor$, that determines the classification of the performance analysis of various module characteristics.

Lemma 3.4.1 $\Lambda(a, i - 1) \subset \Lambda(a, i)$, where $a \geq 1$ and $i > a$. □

This lemma is obvious since $i > i - 1$ and the address cycles are identical.

Theorem 3.4.3 The total number of distinct line states for module characteristics (a, c) with $c > a$ is

$$N(a, c) = N(a, c - 1) + N(a, c - a),$$

where for $1 \leq c \leq a$, $N(a, c) = c$.

Proof: A module which accepts a request goes through all $c - 1$ busy module states sequentially in ascending order. In order to prove the recursive relationship, consider that there are $c - 1$ slots in which an object representing a busy module on a line can be placed. The slots are numbered sequentially from 1 to $c - 1$. An object is placed in slot $r \in \{1, 2, \ldots, c - 1\}$, if a module on that line is busy because it accepted a request r STUs ago. Hence $N(a, i)$ is the number of distinct ways of placing identical objects in $i - 1$ slots such that if there is an object in slot r_i and another in r_j, then $|r_i - r_j| \geq a$. Note that $N(a, i)$ includes the case in which no object is placed in any of the $c - 1$ slots corresponding to all modules on that line being idle.

The memory cycle determines how long a module remains busy hence $\Lambda(a, c) = \{\emptyset, (1), (2), \ldots, (c - 1)\}$, for $1 \leq c \leq a$. Hence for $1 \leq c \leq a$, $N(a, c) = c$. $\Lambda(a, i - 1) \subset \Lambda(a, i)$, for $i > a$, hence $N(a, i) = N(a, i - 1)$ + the number of states due to the effect of increasing the memory cycle by 1. Therefore, consider the effect of adding an $(i - 1)$th slot to the $i - 2$ existing slots. This added slot corresponds to increasing the memory cycle from $i - 1$ to i. All new combinations of placing identical objects in the slots must contain an object in slot $i - 1$. However, the presence of an object in slot $i - 1$ precludes the placement of an object in any of the previous $a - 1$ slots.

Hence the slots that effectively partake in the placement of objects due to the addition of the $(i - 1)$th slot are $1, 2, 3, \ldots, i - 1 - a$. However, the number of distinct ways of placing objects in $i - a - 1$ slots, subject to the rule that no two objects are less than a STU apart, is $N(a, i - a)$. Hence, for $i = c$, $N(a, c) = N(a, c - 1) + N(a, c - a)$. □

Notice that $N(1, c)$ is a binary series in c and $N(2, c)$ is the Fibonacci series. For properties of the general series, see $W(a, 1, c)$ in [10]. It can be easily shown that $N(i, i) = i$ and $N(i, i + 1) = i + 1$, for $i \geq 1$. As an illustration, consider the case of $(a, c) = (2, 6)$. $N(2, 6) = N(2, 5) + N(2, 4)$. $N(2, 5) = N(2, 4) + N(2, 3)$ and $N(2, 4) = N(2, 3) + N(2, 2)$. Hence $N(2, 6) = 3N(2, 3) + 2N(2, 2) = 3x3 + 2x2 = 13$.

There is no known closed form solution for $N(a, c)$ derivable from the recursive relation. However, $N(a, c)$ can be rederived differently.

Theorem 3.4.4 The number of distinct line states with β elements in $\Lambda(a, c)$ is

$$n(\beta) = \sum_{j_{\beta-1}=1}^{c-(\beta-1)a-1} \sum_{j_{\beta-2}=1}^{j_{\beta-1}} \cdots \sum_{j_2=1}^{j_3} \sum_{j_1=1}^{j_2} j_1, \quad \beta \geq 2.$$

$n(0) = 1$, $n(1) = c - 1$.

Proof: It is immediately obtained that $n(0) = 1$ (state $\emptyset$) and $n(1) = c - 1$ (states $(1), (2), \ldots, (c - 1)$). Assume that the elements of a line state are listed in ascending order. Let r_k represent the kth

element of a line state with β elements. Then $r_{k+1} - r_k \geq a$, for $0 < k < \beta$. The first elements, r_1, can take a value between 1 and $c - (\beta - 1)a - 1$ inclusive. Hence the second elements, r_2, can take a value between $r_1 + a$ and $c - (\beta - 2)a - 1$ inclusive. In general, the kth element can take a value between $r_{k-1} + a$ and $c - (\beta - k)a - 1$ for $1 < k \leq \beta$.

Hence,

$$n(\beta) = \sum_{r_1=1}^{c-(\beta-1)a-1} \sum_{r_2=r_1+a}^{c-(\beta-2)a-1} \cdots \sum_{r_{\beta-1}=r_{\beta-2}+a}^{c-a-1} \sum_{r_\beta=r_{\beta-1}+a}^{c-1} 1$$

Let $j_1 = c - a - r_{\beta-1}$. If $r_{\beta-1} = r_{\beta-2} + a$, $j_1 = c - 2a - r_{\beta-2}$. Similarly, if $r_{\beta-1} = c - a - 1$, $j_1 = 1$. With this change of variable, the innermost two summations can be rewritten thus:

$$\sum_{r_{\beta-1}=r_{\beta-2}+a}^{c-a-1} (c-a-r_{\beta-1}) = \sum_{j_1=c-2a-r_{\beta-2}}^{1} j_1 = \sum_{j_1=1}^{c-2a-r_{\beta-2}} j_1$$

In general, let $j_i = c - ia - r_{\beta-i}$, for $2 \leq i \leq \beta - 2$ and $j_{\beta-1} = r_1$. If for $2 \leq i \leq \beta - 2$, $r_{\beta-i} = r_{\beta-i-1} + a$, $j_i = c - (i + 1)a - r_{\beta-i-1} = j_{i+1}$, and if $r_{\beta-i} = c - ia - 1$, $j_i = 1$. Hence applying these changes of variables as in the above example, to $n(\beta)$,

$$n(\beta) = \sum_{j_{\beta-1}=1}^{c-(\beta-1)a-1} \sum_{j_{\beta-2}=1}^{j_{\beta-1}} \cdots \sum_{j_2=1}^{j_3} \sum_{j_1=1}^{j_2} j_1$$

□

Corollary 3.4.4.1 The total number of distinct line states in $\Lambda(a, c)$ is

$$N(a, c) = \sum_{\beta=0}^{\left\lceil \frac{c-1}{a} \right\rceil} n(\beta).$$

□

This result follows from Theorems 3.4.1 and 3.4.4. The following corollaries are results for some specific cases, using the formulas,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad \text{and} \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6},$$

and Theorem 3.4.4.

Corollary 3.4.4.2 For $\left\lceil \frac{c-1}{a} \right\rceil = 1$, $N(a, c) = c$. □

Corollary 3.4.4.3 For $\left\lceil \frac{c-1}{a} \right\rceil = 2$, $N(a, c) = c + \frac{(c-a-1)(c-a)}{2}$. □

Corollary 3.4.4.4 For $\left\lceil \frac{c-1}{a} \right\rceil = 3$, $N(a, c) = c + \frac{(c-a-1)(c-a)}{2} + \frac{(c-2a-1)(c-2a)(c-2a+1)}{6}$. □

The above corollaries are examples of closed form solutions of $N(a, c)$ for three different classes of module characteristics. It is obvious that the magnitude of $N(a, c)$ increases nonlinearly with $\left\lceil \frac{c-1}{a} \right\rceil$.

3.5 Probability of Acceptance, $P_A(a, c, p)$

No closed form solution for $P_A(a, c, p)$ exists for the general module characteristics (a, c), even for $p = 1$. We must know the (a, c) values and obtain the reduced line state graph, $G'_\ell(a, c)$, in order to solve the Markov state diagram for the probability of acceptance, $P_A(a, c, p)$, for $p = 1$. The technique used requires the computation of the steady state

probability of being in an acceptance line state. This method is not readily applicable to the general case for $p \geq 1$.

In this section, the steady state probability of acceptance, $P_A(a, c, p)$ for $p \geq 1$, is discussed for certain classes of module characteristics. Some of the results discussed here were presented in [24]. It is also shown that the complexity of $P_A(a, c, p)$ increases with the maximum number of busy modules on an idle line, $\left\lfloor \frac{c-1}{a} \right\rfloor$ for $a > 1$. However, closed form expressions exist for $P_A(a, c, p)$ in the following cases: $c \geq a = 1$ and $a \leq c \leq 2a$. These classes of module characteristics cover more than 78% of the possible cases of module characteristics, if $c \leq 10$.

In Chapter 2, it was shown that a request can be rejected due to three types of memory collisions. For convenience, the three types of memory collisions are repeated here. A request made to the memory system may be rejected due to

1. MALC: Multiple Access Line Collision (only if $p > 1$),
2. BLC: Busy Line Collision (only if $a > 1$), or
3. BMC: Busy Module Collision (only if $c > a$).

Recall that p requests are issued simultaneously. Hence if more than one request addresses the same line, only one of these may be accepted. Let P_1, P_2 and P_3 be the probabilities of rejection of a request due to MALC, BLC and BMC respectively. Then the probability of a request being rejected by the memory system can be obtained by considering all the mutually exclusive and independent rejection and nonrejection events.

Theorem 3.5.1 The probability of rejection of a request issued to the memory system whose module characteristics are (a, c) is

$$P_R(a, c, p) = P_1 + (1 - P_1)P_2 + (1 - P_1)(1 - P_2)P_3.$$ □

Notice that $1 - P_1$ and $1 - P_2$ are the probabilities that a rejection does not occur (nonrejection) due to MALC and BLC respectively.

Corollary 3.5.1.1 The probability of acceptance of a request made to the memory whose module characteristics are (a, c) is

$$P_A(a, c, p) = 1 - P_R(a, c, p).$$ □

For brevity, $P_A(a, c, p)$ and $P_R(a, c, p)$ will sometimes be written as P_A and P_R respectively. Hence in order to obtain $P_A(a, c, p)$, it is sufficient to know P_1, P_2 and P_3. Notice that $P_A(a, c, p) = (1 - P_1)(1 - P_2)(1 - P_3)$.

Lemma 3.5.1 The probability of a request being rejected due to a multiple access line collision (MALC) with one or more of the $p - 1$ other simultaneous requests is

$$P_1 = 1 - [1 - (\frac{\ell-1}{\ell})^p] \frac{\ell}{p}$$

Proof: There are ℓ^p and $(\ell - 1)^p$ distinct ways of mapping p requests to ℓ and $\ell - 1$ lines respectively. Hence, there are $[\ell^p - (\ell - 1)^p]$ maps which reference a particular line at least once. Thus the expected number of distinct lines referenced by p memory requests; i.e., the line bandwidth is $\frac{[\ell^p - (\ell-1)^p]\ell}{\ell^p}$. The probability, $P_1 = 1 - \frac{\text{line bandwidth}}{p}$. □

It is interesting to note that $1 - P_1$ is a closed form representation of Ravi's results in [18]. Chang showed the equivalence of $1 - P_1$ and Ravi's result in [25].

A request will be rejected due to BLC if it has no MALC, but references a busy line.

Lemma 3.5.2 The probability of a request referencing a busy line is

$$P_2 = \frac{p(a-1)P_A}{\ell} .$$

Proof: The expected number of busy lines which can cause rejection of an incoming request is equal to the expected number of accepted requests in the most recent a - 1 STUs = $p(a - 1)P_A$.

Hence $$P_2 = \frac{p(a-1)P_A}{\ell}$$ □

Corollary 3.5.2.1 The expected number of idle lines is

ℓ_{idle} = ℓ - expected number of busy lines, hence

$\ell_{idle} = \ell - p(a - 1)P_A$, and

$\ell_{idle} = \ell(1 - P_2)$. □

The computation of the probability of referencing a busy module on an idle line, P_3, is not always straightforward. However, P_3 can be generalized by the next lemma.

Lemma 3.5.3 The probability of referencing a busy module on an idle line is

$$P_3 = \frac{E(BM/IL)}{E(M/IL)} ,$$

where E(BM/IL) is the expected number of busy modules on idle lines and E(M/IL) is the expected number of modules on idle lines. □

This formula can be easily derived from simple probability theory. Notice that $E(M/IL) = \ell_{idle} \cdot m = \ell m(1 - P_2) = N(1 - P_2)$.

The derivation of E(BM/IL) for given (a, c) can be made from the reduced line state graphs, $G'_\ell(a, c)$. Notice that previously, the reduced line state graph was used to compute $P_A(a, c, p)$, for $p = 1$. However, E(BM/IL) involves the general case for $p \geq 1$. Since all ℓ lines are identical and independent, the reduced line state graph for all ℓ lines are identical and independent. Hence the transition probability between any two states in one line state graph is identical to the transition probability between the same two states in another line state graph of the same module characteristics, (a, c). At steady state, if p requests are issued, pP_A requests are accepted. These accepted requests cause the addressed lines to make transitions to acceptance line states. Since the request references are uniformly distributed over the ℓ lines, the accepted requests will be uniformly distributed over all ℓ lines and hence over all acceptance line states in all ℓ line state graphs.

For the whole system, let us collapse all ℓ line state graphs into one. The resulting state graph is identical to a line state graph and will therefore be referred to as a line state graph. The following definitions are made to aid in the derivation of E(BM/IL) for the whole system.

Definition 3.5.1 For the whole system, $E(\lambda)$ is the expected number of lines at any time instant which make transitions into line state $\lambda \in G'_{\ell}(a, c)$.

□

Definition 3.5.2 $\Lambda'_1(a, c)$ is the set of nonnull idle line states in $G'_{\ell}(a, c)$. Hence $\Lambda'_1(a, c) = \{\lambda \mid \lambda \in G'_{\ell}(a, c)$, if $r \in \lambda$ then $r \geq a$, $\lambda \neq \emptyset\}$

□

Lemma 3.5.4 $$\sum_{\substack{\lambda \in G'_{\ell}(a, c) \text{ and} \\ \lambda = \text{acceptance state}}} E(\lambda) = pP_A$$

□

This lemma is obvious since accepted requests cause transitions of the addressed lines into acceptance line states.

Lemma 3.5.5 $$E(BM/IL) = \sum_{\lambda \in \Lambda'_1(a,c)} \left(E(\lambda) \cdot |\lambda|\right).$$

□

This lemma follows from the definition of E(BM/IL). The expected number of busy modules on idle lines is the expected number of busy modules on lines which make transitions to nonnull idle line states.

For the case a = 1, some further results are easily obtained. In this case, the busy modules are uniformly distributed over the ℓ lines and no lines are busy. Hence $\ell_{idle} = \ell$ at all times.

Lemma 3.5.6 For a = 1, the probability of a request referencing a busy module is

$$P_3 = \frac{p(c - 1)}{N} P_A$$

Proof: Since the busy modules are uniformly distributed on the lines for a = 1, the expected number of busy modules which can cause rejection of an incoming request is equal to the expected number of accepted requests in the most recent c - 1 STUs = $p(c-1)P_A$. Since there are a total of N memory modules in the system,

$$P_3 = \frac{p(c - 1)P_A}{N} .$$ □

For the special case a = c = 1, $P_2 = P_3 = 0$. Hence $P_R(a, c, p) = P_1$ and $P_A(a, c, p) = 1 - P_1 = [1 - (\frac{\ell-1}{\ell})p] \frac{\ell}{p}$. This is the only case considered by previous investigators [19, 20, 21, 22]. However, for the case $c \geq a = 1$, the probability of acceptance can be obtained from the following theorem.

Theorem 3.5.2 For a = 1, $c \geq 1$, the probability of acceptance, $P_A(a, c, p)$ of any (ℓ, m) memory configuration is

$$P_A(1, c, p) = \frac{1 - P_1}{1 + (1-P_1)k_2}$$

where $k_2 = \frac{p(c - 1)}{N}$.

Proof: To prove this, note that since a = 1, there will be no rejections due to busy lines. The probability of rejection, P_R, of the (ℓ, m) memory organization with module characteristics (a = 1, c > 1) can now

be easily obtained since rejection can only occur due to a multiple access line collision or a busy module request:

$$P_R = 1 - P_A = P_1 + (1 - P_1) \cdot P_3$$

and $$1 - P_A = P_1 + (1 - P_1) \frac{p(c - 1)P_A}{N} .$$

Hence $P_A(1, c, p) = \frac{1 - P_1}{1 + (1 - P_1) \cdot k_2}$, where $k_2 = \frac{p(c - 1)}{N}$. □

For the special case of $p = 1$, $a = 1$, $P_1 = 0$, and $P_A(1, c, 1) = \frac{1}{1 + k_2} = \frac{N}{N + c - 1}$. Thus for $p = 1$, the performance of a memory with $a = 1$ is a function of $N(= \ell m)$ and c only and not of the (ℓ, m) memory configuration.

Notice that the line state graph was not required to obtain P_A for $a = 1$. For $a = 1$, the lines are always idle at any time instant and are therefore ready to accept a request provided the request is not made to a busy module on the line. This simplifies the problems.

Computing P_3 for $a > 1$ is not an easy task since the busy modules are not uniformly distributed on the ℓ lines, although the requests are uniformly distributed over the ℓ lines. However, P_3 can be calculated for certain classes of (a, c) by classifying the line state graphs. The transition probabilities in a line state graph are either $(m - |\lambda|)N$, $1 - (m - |\lambda|)/N$ or 1. For an idle line state, $\lambda \in \Lambda_I(a, c)$, $|\lambda| \leq \left\lfloor \frac{c - 1}{a} \right\rfloor$, hence the lower limit of the probability of transition from an idle line state, $\lambda \neq \emptyset$, to its successor acceptance state is $(m - \left\lfloor \frac{c - 1}{a} \right\rfloor)/N$. Hence, for $a > 1$, the probability of transition from idle line state, λ, to $f_a(\lambda)$ in $G'_\ell(a, c)$ is $\alpha_j = (m - j)/N$ where $j = |\lambda|$ and

$0 \leq j \leq \left\lfloor \frac{c-1}{a} \right\rfloor$. Consider the case where $\left\lfloor \frac{c-1}{a} \right\rfloor = 1$, for $a > 1$.

Theorem 3.5.3 For $a \leq c \leq 2a$, $a > 1$, the probability of acceptance, $P_A(a, c, p)$ of any (ℓ, m) memory configuration is given by

$$P_A(a, c, p) = \frac{1 - P_1}{1 + (1-P_1)k_1 + \frac{p(1-P_1)}{m-1}\left[1 - (1-\alpha_1)^{c-a}\right]},$$

where $k_1 = \frac{p(a-1)}{\ell}$ and $\alpha_1 = \frac{m-1}{N} > 0$.

Also, $P_A(a, c, p) = \frac{1 - P_1}{1 + (1-P_1)k_2}$ for $\alpha_1 = 0$, where $k_2 = \frac{p(c-1)}{N}$.

Proof: For $a < c \leq 2a$ and $a > 1$, $\left\lfloor \frac{c-1}{a} \right\rfloor = 1$. Hence there is exactly one busy module on a nonnull idle line state in $\Lambda(a, c)$. Figure 3.5.1 shows the line state graph, $G_\ell(a, c)$, for $a < c \leq 2a$, $a > 1$. It can easily be seen that each busy line state, λ, with two elements is equivalent to a busy line state with one element r, such that r is the smallest element in λ. For example, $(1, a+1) \sim (1, a+2) \sim (1, a+3) \sim \ldots \sim (1, c-1) \sim (1)$. Hence such states can be merged which results in the reduced line state graph $G'_\ell(a, c)$ for $a < c \leq 2a$, $a > 1$ shown in figure 3.5.2. The probability of transition from a nonnull idle line state to the acceptance state, (1), is $\alpha_1 = (m-1)/N$.

The expected number of busy modules on idle lines, E(BM/IL), can be obtained from $G'_\ell(a, c)$. The expected number of accepted requests every STU is pP_A. Let $E(\lambda)$ denote the expected number of lines which make transitions into line state λ. Hence $E(1) = pP_A$,

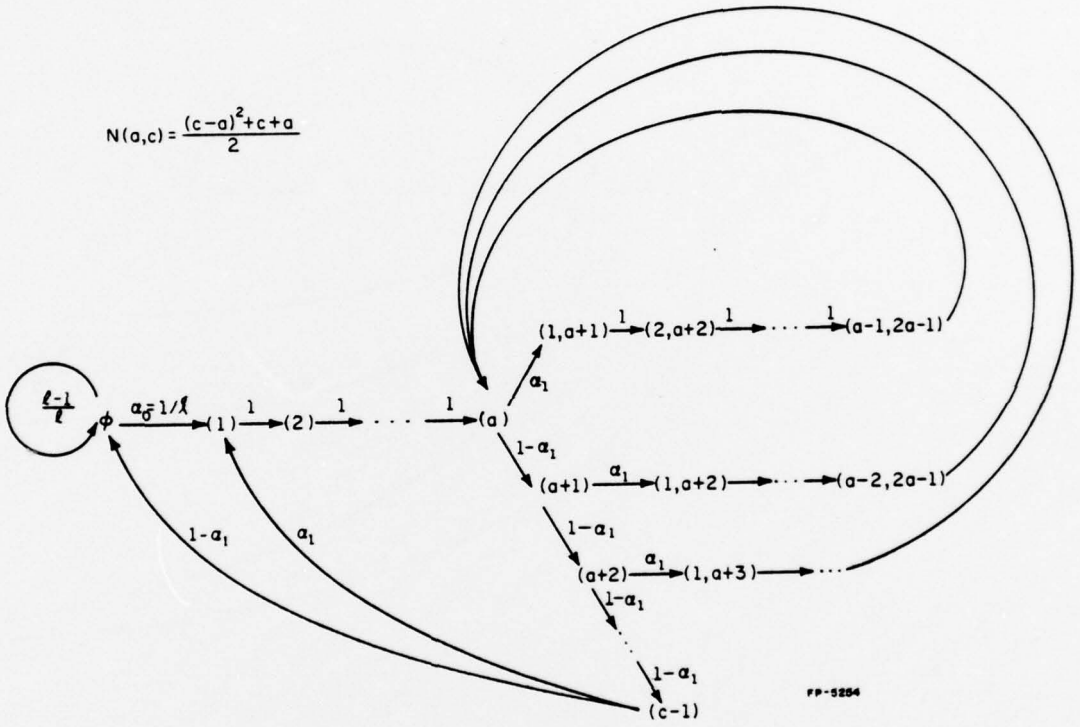


Figure 3.5.1 Line State Graph, $G_\ell(a, c)$, for $a < c \leq 2a$ and $a > 1$

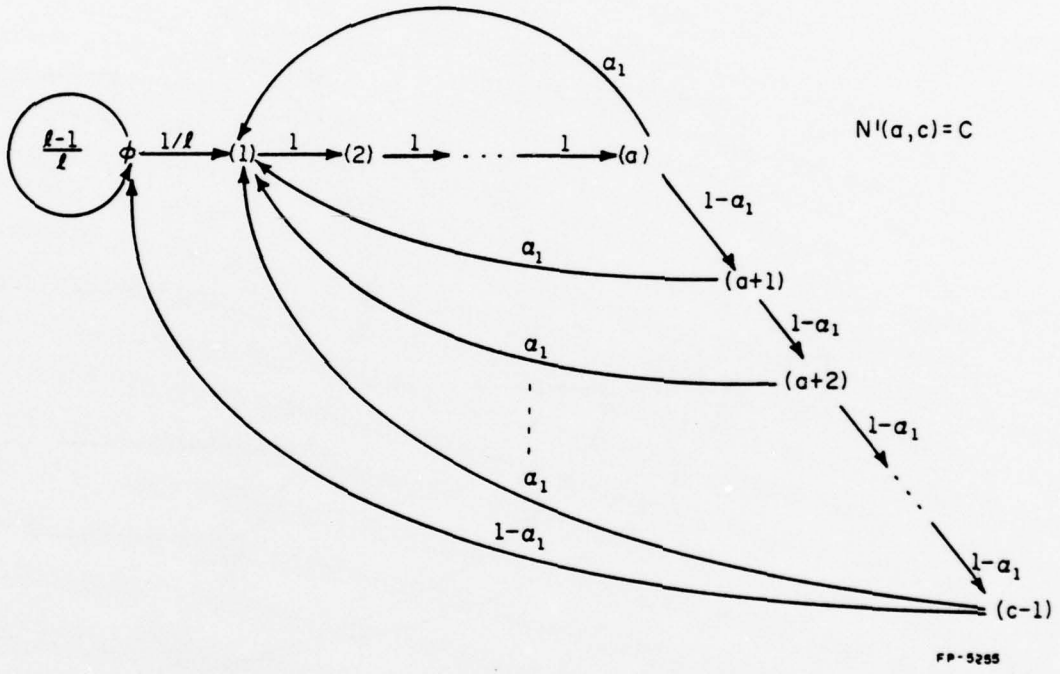


Figure 3.5.2 Reduced Line State Graph, $G^1_\ell(a, c)$ for $a < c \leq 2a$ and $a > 1$

since $\lambda = (1)$ is the only acceptance state in $G'_{\ell}(a, c)$. Let P_{λ} denote the steady state probability of being in state λ. From $G'_{\ell}(a, c)$ in figure 3.5.2, $P_{(1)} = P_{(2)} = \ldots = P_{(a)}$, since the transition probability from state (r) to (r + 1) is 1 for $1 \leq r \leq a - 1$. Hence

$$E(1) = E(2) = \ldots = E(a) = pP_A.$$

The states that represent busy modules on idle lines are (a), (a + 1), ..., (c - 2) and (c - 1), each of which has one busy module. Hence

$$E(BM/IL) = \sum_{i=a}^{c-1} E(i).$$

Note that $\lambda = \emptyset$ does not represent a busy module on an idle line, hence it is not included in E(BM/IL). From the nonacceptance transition of nonnull idle line states,

$$E(a+1) = (1-\alpha_1)\, E(a),\ E(a+2) = (1-\alpha_1)\, E(a+1),\ \ldots$$

$$\ldots\ E(c-1) = (1-\alpha_1)\, E(c-2).$$

Hence in general, $E(r) = (1 - \alpha_1)\, E(r - 1)$, for $a + 1 \leq r \leq c - 1$.

Thus, $E(r) = (1 - \alpha_1)^{r-a}\, E(a)$, for $a \leq r \leq c - 1$. Therefore

$$E(BM/IL) = \sum_{i=a}^{c-1} (1-\alpha_1)^{i-a}\, E(a).$$

Since $E(a) = pP_A$,

$$E(BM/IL) = pP_A \sum_{i=a}^{c-1} (1-\alpha_1)^{i-a} = pP_A \sum_{i=0}^{c-a-1} (1-\alpha_1)^{i}$$

$$= \frac{pP_A}{\alpha_1} [1-(1-\alpha_1)^{c-a}],\ \alpha_1 > 0.$$

From previous discussion, $E(M/IL) = N(1 - P_2)$. Hence

$$P_3 = \frac{pP_A[1-(1-\alpha_1)^{c-a}]}{N\alpha_1(1-P_2)},\ \alpha_1 > 0.$$

Therefore $P_R = 1-P_A = P_1 + (1-P_1)P_2 + \dfrac{(1-P_1)\cancel{(1-P_2)}pP_A[1 - (1-\alpha_1)^{c-a}]}{\cancel{(1-P_2)}\, N\alpha_1},\ \alpha_1>0.$

Substituting for $P_2 = \frac{p(a-1)P_A}{\ell}$ and manipulating,

$$1-P_1 = P_A\{1 + \frac{(1-P_1)\ p(a-1)}{\ell} + \frac{(1-P_1)\ p[1-(1-\alpha_1)^{c-a}]}{N\alpha_1}\},$$

where $\alpha_1 = \frac{m-1}{N} > 0$. Letting $k_1 = \frac{p(a-1)}{\ell}$ and solving for P_A, the theorem is proved for $\alpha_1 > 0$.

However, when $\alpha_1 = 0$, $m = 1$ and $\ell = N$, figure 3.5.3 shows the resulting reduced line state graph. In this case, it is easy to show that $E(BM/IL) = p(c-a)P_A$. Hence,

$$P_3 = \frac{p(c-a)P_A}{N(1-P_2)}, \quad \alpha_1 = 0 .$$

Substituting for P_2 and P_3, $P_A(a, c, p) = \frac{1-P_1}{1+(1-P_1)k_2}$, where $k_2 = \frac{p(c-1)}{N}$.

For $a = c$, $P_3 = 0$, hence $P_A(a, c, p) = \frac{(1-P_1)}{1+(1-P_1)k_1}$, which is also derivable from the two results above for $\alpha_1 > 0$ and $\alpha_1 = 0$ by substituting $a = c$ in the equations. □

It is seen that the solution of $P_A(a, c, p)$ for $a \leq c \leq 2a$, $a > 1$ is fairly involved. There is no known general solution of $P_A(a, c, p)$ for general module characteristics. The next higher class of module characteristics presents a more complex state graph. This is the case for $\left\lfloor \frac{c-1}{a} \right\rfloor = 2$, where $2a < c \leq 3a$ and $a > 1$. An exact soiution for $P_A(a, c, p)$ was not obtained because of the complexity involved. Instead an approximate solution for large m was favored, although this is still relatively complex. The complexity arises in obtaining P_3. Notice that for large m, $\alpha_1 \approx \alpha_2$. Hence assume that $\alpha_1 \approx \alpha_2 = \alpha$.

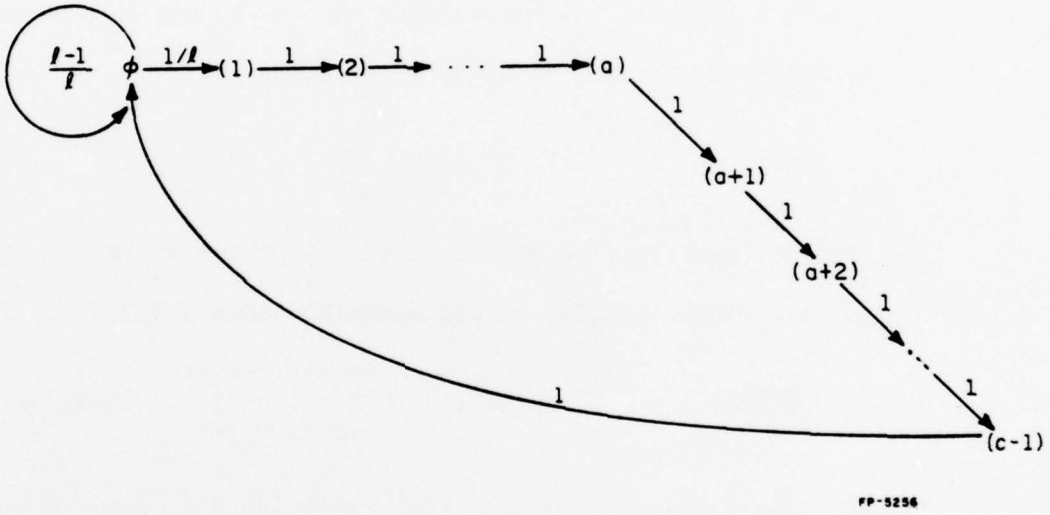


Figure 3.5.3 Reduced Line State Graph, $G'_{\ell}(a, c)$, for $a < c \leq 2a$ and $a > 1$, where $m = 1$ and $\ell = N$

Theorem 3.5.4 If $\alpha_1 \approx \alpha_2$, and (a, c) is such that $2a < c \leq 3a$, $a > 1$,

$$P_A(a, c, p) = \frac{1 - P_1}{1 + (1-P_1)k_1 + \frac{p(1-P_1)}{N\alpha}\left[1 - (1-\alpha_1)^{c-a} + k_2\right]},$$

for $\alpha \neq 0$, where $k_1 = \frac{p(a-1)}{\ell}$ and $k_2 = 1 - [1 + \alpha(c-a)]\ (1-\alpha)^{c-2a}$.

Proof: Figure 3.5.4 illustrates the reduced state graph, $G'_\ell(a, c)$ for $2a < c \leq 3a$. An example is shown for the (a, c) = (2, 6) case in figure 3.5.5. In the following analysis, α_1 and α_2 are replaced by α. Applying lemma 3.5.4 to $G'_\ell(a, c)$ of figure 3.5.4,

$$E(1) + \sum_{i=0}^{c-2a-1} E(1, a+i+1) = pP_A \quad \ldots\ldots\ldots\ldots\ldots\ldots\ 1.$$

From an investigation of the idle line states representing the busy modules on idle lines in $G'_\ell(a, c)$ and applying lemma 3.5.5,

$$E(BM/IL) = \sum_{i=0}^{c-a-1} E(a+i) + \sum_{i=0}^{c-2a-1} \sum_{j=0}^{c-2a-i-1} E(a+j, 2a+j+i) \quad \ldots\ldots\ 2.$$

But $E(a+j, 2a+j+i) = (1-\alpha)^j E(a, 2a+i)$ and $E(a, 2a+i) = E(1, a+i+1)$, hence substituting and using the relation $\sum_{j=0}^{n-1} r^i = \frac{1 - r^n}{1 - r}$,

$$E(BM/IL) = \sum_{i=0}^{c-a-1} E(a+i) + \sum_{i=0}^{c-2a-1} \frac{1 - (1-\alpha)^{c-2a-i}}{\alpha} E(1, a+i+1) \quad \ldots\ 3.$$

Considering the single element idle line states in the state graph,

$$E(a+i+1) = (1-\alpha)\ E(a+i) + (1-\alpha)\ E(a+i, c-1), \text{ for}$$

$$0 \leq i \leq c-2a-1 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ 4.$$

and $E(a+i+1) = (1-\alpha)^{i-(c-2a-1)}\ E(c-a)$, for $c-2a-1 \leq i \leq c-a-2$.

This can be rewritten as

$$E(c-a+i) = (1-\alpha)^i\ E(c-a),\ 0 \leq i \leq a-1.$$

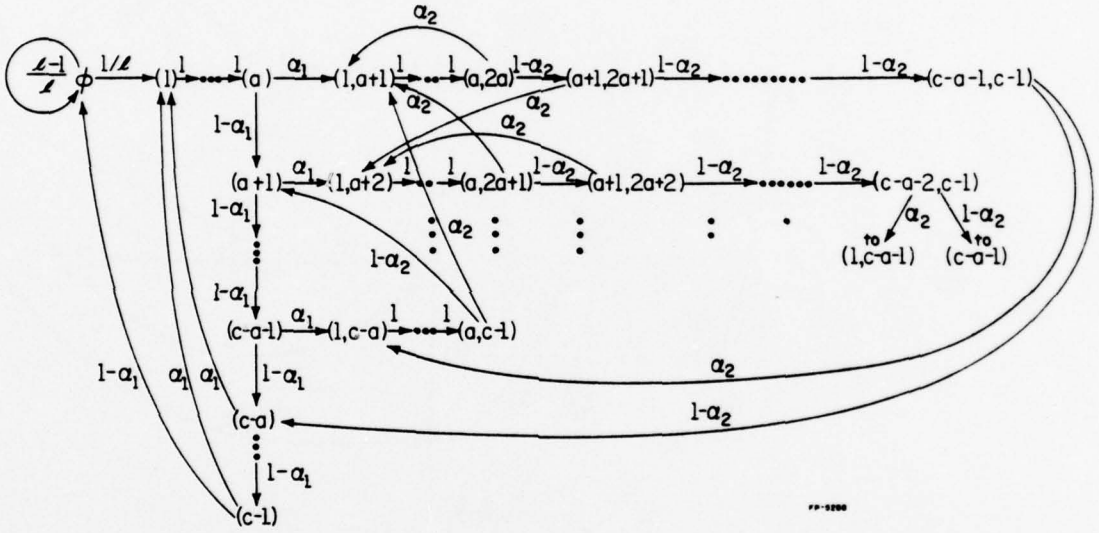


Figure 3.5.4 Reduced Line State Graph, $G'_{\ell}(a, c)$, for $2a < c \leq 3a$ and $a > 1$

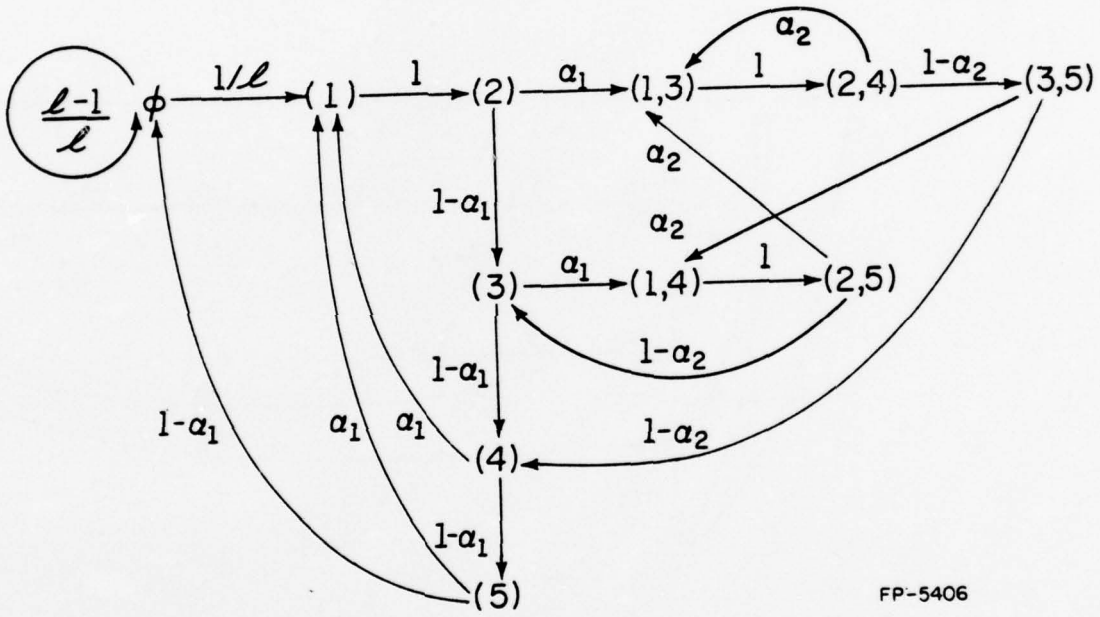


Figure 3.5.5 Reduced Line State Graph, $G'_\ell(2, 6)$, where $\alpha_j = \frac{m-j}{N}$

Hence

$$\sum_{i=c-2a-1}^{c-a-2} E(a+i+1) = \sum_{i=0}^{a-1} E(c-a+i) = \sum_{i=0}^{a-1} (1-\alpha)^i E(c-a)$$

$$\therefore \sum_{i=c-2a-1}^{c-a-2} E(a+i+1) = \frac{1 - (1-\alpha)^a}{\alpha} E(c-a) \quad \ldots\ldots\ldots\ldots\ldots\ldots 5.$$

From the nonacceptance of two-element states,

$$E(a+i, c-1) = (1-\alpha)^i E(1, c-a-i),\ 0 \leq i \leq c-2a-1 \quad \ldots\ldots\ldots 6.$$

Substituting for $E(a+i, c-1)$ in equation 4,

$$E(a+i+1) = (1-\alpha)[E(a+i) + (1-\alpha)^i E(1, c-a-i)]. \quad \ldots\ldots\ldots\ldots 7.$$

It can be shown from equation 7 that in general,

$$E(a+i) = (1-\alpha)^i \left[E(a) + \sum_{j=c-2a-i}^{c-2a-1} E(1, a+j+1)\right],\ 1 \leq i \leq c-2a \quad \ldots 8.$$

Considering the transitions to the two-element acceptance states, for $0 \leq i \leq c-2a-1$,

$$E(1, a+i+1) = \alpha\left[E(a+i) + \sum_{j=0}^{c-2a-i-1} E(a+i, 2a+i+j)\right] \quad \ldots\ldots\ldots 9.$$

But, $E(a+i, 2a+i+j) = (1-\alpha)^i E(a, 2a+j) = (1-\alpha)^i E(1, a+j+1)$,

hence, substituting for $E(a+i, 2a+i+j)$ in equation 9,

$$E(1, a+i+1) = \alpha\left[E(a+i) + \sum_{j=0}^{c-2a-i-1} (1-\alpha)^i E(1, a+j+1)\right] \quad \ldots\ldots 10.$$

Substituting for $E(a+i)$ from equation 8,

$$E(1, a+i+1) = \alpha(1-\alpha)^i\left[E(a) + \sum_{j=0}^{c-2a-1} E(1, a+j+1)\right] \quad \ldots\ldots\ldots 11.$$

Since $E(a) = E(1)$ and from equation 1,

$$E(1, a+i+1) = \alpha(1-\alpha)^i pP_A,\ 0 \leq i \leq c-2a-1. \quad \ldots\ldots\ldots\ldots 12.$$

The first term of equation 3 can be rewritten as

$$\sum_{i=0}^{c-a-1} E(a+i) = \sum_{i=0}^{c-2a-1} E(a+i) + \sum_{i=c-2a}^{c-a-1} E(a+i) \quad \ldots\ldots\ldots\ldots 13.$$

From equation 8,

$$E(c-a) = (1-\alpha)^{c-a} \left[E(a) + \sum_{j=0}^{c-2a-1} E(1,\ a+j+1)\right]$$

Recognizing the bracketed term as pP_A,

$$E(c-a) = (1-\alpha)^{c-a}\ pP_A \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 14.$$

From equations 1 and 12,

$$E(1) + \alpha pP_A \sum_{i=0}^{c-2a-1} (1-\alpha)^i = pP_A$$

Since $E(1) = E(a)$, evaluating the above expression,

$$E(1) = E(a) = (1-\alpha)^{c-2a}\ pP_A \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots 15.$$

From equations 8, 12, and 15,

$$E(a+i) = (1-\alpha)^i \left[(1-\alpha)^{c-2a}\ pP_A + \sum_{j=c-2a-i}^{c-2a-1} \alpha(1-\alpha)^j\ pP_A\right]$$

$$E(a+i) = (1-\alpha)^i\ pP_A \left[(1-\alpha)^{c-2a} + (1-\alpha)^{c-2a-i} - (1-\alpha)^{c-2a}\right]$$

$$E(a+i) = (1-\alpha)^{c-2a}\ pP_A,\ 0 \leq i \leq c-2a \quad \ldots\ldots\ldots\ldots\ldots 16.$$

Combining equations 5, 14, and 16, equation 13 become

$$\sum_{i=0}^{c-a-1} E(a+i) = \sum_{i=0}^{c-2a-1} (1-\alpha)^{c-2a}\ pP_A + \frac{1-(1-\alpha)^a}{\alpha} \cdot (1-\alpha)^{c-a}\ pP_A$$

$$\sum_{i=0}^{c-a-1} E(a+i) = \left[(c-2a) + \frac{1-(1-\alpha)^a}{\alpha}\right] (1-\alpha)^{c-2a}\ pP_A \quad \ldots\ldots\ldots 17.$$

Substituting for E(1, a+i+1) from equation 12 into last term of equation 3,

$$\sum_{i=0}^{c-2a-1} \frac{1 - (1-\alpha)^{c-2a-i}}{\alpha} \cdot \alpha(1-\alpha)^i \, pP_A = 2pP_A\left[\sum_{i=0}^{c-2a-1} (1-\alpha)^i - \sum_{i=0}^{c-2a-1} (1-\alpha)^{c-2a}\right]$$

$$= 2\left[\frac{1 - (1\alpha)^{c-2a}}{\alpha} - (c-2a)\,(1-\alpha)^{c-2a}\right] pP_A \quad \ldots\ldots\ldots\ldots 18.$$

Combining equations 17 and 18,

$$E(BM/IL) = \frac{2[1 - (1-\alpha)^{c-2a}] + \{[1-(1-\alpha)^a] - \alpha(c-2a)\}(1-\alpha)^{c-2a}}{\alpha} pP_A$$

Therefore,

$$E(BM/IL) = \frac{1 - (1-\alpha)^{c-a}}{\alpha} pP_A + \frac{1-[1+\alpha(c-2a)]\,(1-\alpha)^{c-2a}}{\alpha} pP_A \quad \ldots 19.$$

Hence similar to the proof of Theorem 3.5.3,

$$P_A(a, c, p) = \frac{1 - P_1}{1 + (1-P_1)\,k_1 + \frac{p(1-P_1)}{N\alpha}\,[1 - (1-\alpha)^{c-a} + k_2]}, \text{ for } \alpha \neq 0$$

where $k_1 = \frac{p(a-1)}{\ell}$ and $k_2 = 1 - [1 + \alpha(c-2a)](1-\alpha)^{c-2a}$. □

For $\alpha = 0$, a good approximation is the result of Theorem 3.5.3 for $\alpha_1 = 0$, especially for large N.

In general, the complexity of the analysis increases with $\left\lfloor \frac{c-1}{a} \right\rfloor$.

3.6 Bounds on $P_A(a, c, p)$

It was seen in the last section that obtaining $P_A(a, c, p)$ for general module characteristics is a formidable problem. However, upper

and lower bounds can be obtained for $P_A(a, c, p)$ which give a rough estimate for design purposes.

Theorem 3.6.1 The maximum performance memory configuration for any (a, c) is $(\ell, m) = (N, 1)$ and for this configuration,

$$P_A(a, c, p) = \frac{1 - P_1}{1 + (1-P_1)k}, \text{ where } k = \frac{p(c-1)}{N}.$$

Proof: It is trivial to show, since increasing ℓ cannot decrease performance, that the maximum performance memory configuration is $(\ell, m) = (N, 1)$. In this case, since $\ell = N$, we can use the state transition diagram of figure 3.5.3 to evaluate E(BM/IL). Figure 3.5.3 is repeated in figure 3.6.1 for convenience. Notice that when $\ell = N$, $m = 1$ and a line cannot accept a request unless the module on that line is idle. From figure 3.6.1, $E(BM/IL) = \sum_{i=a}^{c-1} E(i) = (c-a)\,E(a)$, since $E(i) = E(i+1)$ for $1 \leq i \leq c-2$. $pP_A = E(1) = \ldots = E(a)$. Therefore, $E(BM/IL) = p(c-a)P_A$. Hence,

$$P_3 = \frac{p(c-a)P_A}{N(1-P_2)}.$$

Using Theorem 3.5.1 and substituting for $P_2 = \frac{p(a-1)P_A}{\ell}$ and $\ell = N$,

$$P_A = \frac{1 - P_1}{1 + (1-P_1)k}, \text{ where } k = \frac{p(c-1)}{N}. \qquad \square$$

Notice that this result is independent of a. It is intuitively obvious that $P_A(a, c, p)$ is a monotonically nondecreasing function in ℓ. Although $(\ell, m) = (N, 1)$ is the maximum performance configuration, this configuration is often undesirable for large N because of the cost incurred by increasing ℓ. Bounds on $P_A(a, c, p)$ for the general memory configuration will be considered.

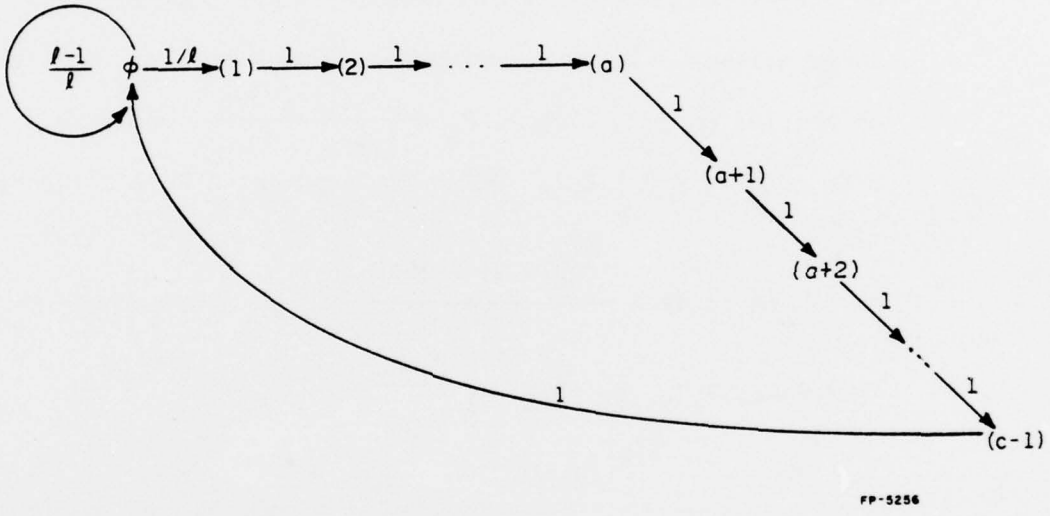


Figure 3.6.1 Reduced Line State Graph, $G'_{\ell}(a, c)$, where $\ell = N$ and $m = 1$

Theorem 3.6.2 For any (a, c) and $p \geq 1$,

$$P_A(a, c, p) \geq \frac{1 - P_1}{\Delta}$$

where $\Delta = 1 + (1-P_1)(k_1+k_2)$, $k_1 = \frac{p(a-1)}{\ell}$ and $k_2 = \frac{p(c-a)}{N}$.

Proof: It is easy to show that at any time instant, the expected number of busy modules is $p(c - 1)P_A$. Of these, $p(a - 1)P_A$ are clearly on busy lines and of the remaining $p(c - a)P_A$ busy modules, let β be on busy lines, where $\beta \geq 0$. Thus there are $p(c-a)P_A - \beta$ busy modules on idle lines. Since P_3 = (number of busy modules on idle lines)/(number of modules on idle lines), $P_3 = \frac{p(c-a)P_A - \beta}{\ell_{idle} \cdot m}$, where $\ell_{idle} = \ell(1 - P_2)$, from corollary 3.5.2.1. Using the expression P_2 and Theorem 3.5.1.

$$P_A(a, c,p) = \frac{1-P_1 + (1-P_1)\beta/N}{\Delta}.$$

Therefore, $P_A(a, c, p) \geq \frac{1 - P_1}{\Delta}$. □

Similarly, an upper bound can be obtained for any (a, c) and $p \geq 1$.

Theorem 3.6.3 For any (a, c) and $p \geq 1$, $P_A(a, c, p) \leq \frac{1 - P_1}{1 + (1-P_1)k_1}$, where $k_1 = \frac{p(a - 1)}{\ell}$.

Proof: From the proof of Theorem 3.6.2,

$$P_A(a, c, p) = \frac{1-P_1 + (1-P_1)\beta/N}{1 + (1-P_1)(k_1+k_2)}$$

where $0 \leq \beta < \infty$, $k_1 = \frac{p(a - 1)}{\ell}$ and $k_2 = \frac{p(c - a)}{N}$. The maximum $P_A(a, c, p)$ occurs when $N = \infty$. In this case, $\frac{\beta}{N} = k_2 = 0$. Hence,

$$P_A(a, c, p) = \frac{1 - P_1}{1 + (1-P_1)k_1} .$$

Therefore,

$$P_A(a, c, p) \leq \frac{1 - P_1}{1 + (1-P_1)k_1} . \qquad \square$$

Hence for any (a, c) and $p \geq 1$, $\frac{1-P_1}{\Delta} \leq P_A(a, c, p) \leq \frac{1 - P_1}{1 + (1-P_1)k_1}$.

Thus for very large N, the effect of referencing a busy module on an idle line is negligible.

Let us denote the upper and lower bounds of $P_A(a, c, p)$ by $P_A(U)$ and $P_A(L)$ respectively. Hence,

$$P_A(U) = \frac{1 - P_1}{1 + (1-P_1)k_1}, \qquad P_A(L) = \frac{1 - P_1}{1 + (1-P_1)(k_1+k_2)} ,$$

where $k_1 = \frac{p(a - 1)}{\ell}$ and $k_2 = \frac{p(c - a)}{N}$. Then, $\frac{1}{P_A(L)} - \frac{1}{P_A(U)} = k_2$.

Hence

$$\frac{P_A(U) - P_A(L)}{P_A(L)} = k_2 \, P_A(U).$$

Therefore,

$$\frac{P_A(U) - P_A(L)}{P_A(L)} \times 100\% = 100k_2 \, P_A(U) = D\%$$

Since $P_A(U) \leq 1$, $D \leq 100k_2 = \frac{100p(c-a)}{N}$, which is independent of the configuration. Notice that for small ℓ, $P_A(U) << 1$, hence $D << 100k_2$. For example, if (a, c) = (3, 8), p = 8 and N = 1024, then D < 4%. Since Since $k_2 = \frac{p(c - a)}{N}$, D will be very small for large N. Hence for large N, the lower bound may serve as a good estimation of $P_A(a, c, p)$.

However, another useful upper bound will be presented.

Theorem 3.6.4 For any (a, c) and $p \geq 1$, $P_A(a, c, p) \leq \min(1, \frac{\ell}{ap})$.

Proof: In a successive STUs, the expected number of accepted requests is apP_A. Since these accepted requests must refer to distinct lines, $apP_A \leq \ell$. Thus $P_A \leq \frac{\ell}{ap}$. Since $P_A \leq 1$, the theorem follows. □

This upper bound will be useful in evaluating the effectiveness of buffering the requests which will be discussed in the next two chapters.

4. SIMULATION OF BUFFERED AND NONBUFFERED REQUESTS

4.1 Introduction

In the previous chapter, it was assumed that rejected requests were discarded so that the independence and randomness over a uniform distribution assumption could be justified. Despite these assumptions, it was seen that the complexity of the Markov model increased with $\left\lfloor \frac{c-1}{a} \right\rfloor$ and it become very difficult to obtain analytic results for classes of (a, c) such that $\left\lfloor \frac{c-1}{a} \right\rfloor \geq 2$.

In this chapter, two practical cases of request schemes will be investigated. In practice, rejected requests are not discarded. Hence one case to be investigated is the nonbuffered request processor (NRP) system, in which rejected requests are not discarded but resubmitted later. The other case is the buffered request processor (BRP) system, in which requests are buffered before being selected for service. Such practical cases are very difficult to model analytically, hence no analytical results have been obtained. However, experimental modeling to evaluate the effectiveness of buffering requests and resubmission of rejected requests on performance are investigated in this chapter.

Simulation of the NRP and BRP systems will be used to obtain experimental results. The effect of the following parameters on performance will be investigated.

(a) Read and write relative module characteristics (a_r, c_r) and (a_w, c_w) respectively

(b) Memory configuration (ℓ, m)

(c) Processor order (s, p)

(d) Memory size, N

There are two kinds of memory requests issued namely, read and write requests. A read request takes a_r and c_r segment time units to complete its address and memory cycles, respectively. Similarly a write request takes a_w and c_w segment time units to complete its address and memory cycles, respectively. It is assumed in the simulation model that the proportion of read requests to write requests is 2:1. Hence the effective module characteristics are (a_e, c_e), where $a_e = 2/3\ a_r + 1/3\ a_w$ and $c_e = 2/3\ c_r + 1/3\ c_w$. For the case studies, it is further assumed that the read address cycle is equal to the write address cycle. Hence $a_e = a_r = a_w$.

The processor system consists of p parallel pipelined processors, where each processor is divided into three units namely, the preprocessor or address generation unit (AGU), the process state unit (PSU) and the processor or execution unit (EXU). Each AGU consists of one segment and hence takes one segment time unit (STU) to complete its processing step.

The PSU consists of c_r segments in series and acts like a shift register, where c_r is the read memory cycle. Each segment contains the process state vector of a distinct process. The process state vector may contain such information as the request status, function, address, and priority. Since each process state vector of a process traverses all c_r segments, the PSU takes c_r STUs to complete its processing step.

The EXU consists of two segments in series and takes two STUs to complete its execution step. It should be pointed out that the total

number of segments in the EXU and AGU can be increased with little or no difference in the simulation results. Therefore, the instruction cycle, $s = c_r + 3$, is fixed for a given pair of module characteristics. If the first segment contains the ith process state vector, then the jth segment contains the process state vector corresponding to the $(i + j - 1) \bmod (c_r + 3)$ process. The contents of the shift register are shifted and updated once per STU.

The AGU generates requests, their addresses and request functions, namely, read or write. The following assumptions are made regarding request arrivals to the memory system.

(1) The arrival time distribution is constant with p requests issued at the beginning of every STU.

(2) Each new request chooses its address from a uniform distribution from 0 to N-1.

Once a request is issued, it stays in the processor memory system until serviced to completion. At the completion of service, a request is terminated and ceases to exist in the system.

Read requests, which are typically instruction or operand fetch commands, require postprocessing, whereas write requests, which are typically store commands, do not require postprocessing. Hence after an accepted read request fetches the instruction or operand from the memory, it is operated on by the segments of the EXU for two STUs. The completion of the execution in the EXU terminates the servicing of the accepted request.

On the other hand, an accepted write request takes one write memory cycle, c_w STUs, to complete its service in the memory. Usually, $c_w \geq c_r$.

In order to simplify the sequencing problem of the simulation model, it was assumed that $c_w = c_r + 2$. Thus simultaneous read and write requests which are accepted by the memory system will eventually terminate their services concurrently. That is, while the accepted read request is about to begin its execution in the first segment of the EXU, the write request, which was accepted simultaneously with the read request, will have exactly two more STUs to complete its memory cycle. Moreover, this sequencing is consistent with a fixed instruction cycle assumption for read and write instruction cycles. Furthermore, it also provides for synchronous cycling of the pipelined processor.

It should be noted that the EXU of each processor retains the process states of processes currently being executed. Furthermore, it typically serves two additional functions, namely, as an instruction decoder and as an instruction execution unit. Hence read requests which are instruction fetches, may use the EXU to decode the instruction. Similarly, a read request which is an operand fetch may use the EXU to execute the instruction.

The p simultaneous requests with their addresses are processed in the accept/reject logic to determine which of them are acceptable requests. That is, the accept/reject logic determines which of the p requests reference idle modules on idle lines. These acceptable requests are then subjected to MALC tests in the priority network to resolve multiple access line collisions which are due to one or more acceptable requests referencing the same line. The output of this test is the set of accepted requests which reference distinct idle lines and idle modules.

Since we are interested mainly in the overall performance of the processor memory system and not in the relative performance of one processor over another, the service discipline becomes irrelevant. However, for experimental purposes, a simple priority scheme is devised for the service discipline. This scheme assigns priorities to different processors such that processor i has priority over processor j, for $i < j$, in accessing a line whenever a multiple access line collision occurs between requests from processors i and j. This priority scheme does not affect the total system throughput. However, other priority schemes may be desirable in practice. The overall performance results are still applicable.

Subsequently, the accepted requests are dispatched, through a $p \times \ell$ crossbar switch, to their respective addressed modules. It is assumed that it takes zero STU to perform the above preacceptance routines. One or two STUs could have been assigned to processing these routines. The effect would be to increase the instruction cycle, s. This increase may affect the transient probability of acceptance of a request in the NRP system because a rejected request is resubmitted s STUs later. However, it would have little or no effect on the steady state probability of acceptance of a request.

The performance of a parallel-pipelined processor of order (s, p) having access to an (ℓ, m) memory configuration with effective module characteristics (a_e, c_e), is based on the following parameters:

(1) The probability of acceptance, P_A (a, c, p)

(2) The expected average wait time of a request from arrival to acceptance.

The above performance evaluators are applicable to both NRP and BRP systems. In addition, for the buffered system, we also consider

(3) The expected average queue length.

The probability of acceptance is the probability that a typical request which arrives at the memory will be accepted. The throughput of the processor memory system is proportional to the probability of acceptance, when the number of processes which request memory within one effective memory cycle is held fixed and the memory cycle is fixed too.

Since fixed priorities are assigned to processors, a processor with a higher priority will exhibit lower expected wait time and smaller expected queue length. In order to take into account the effect of the priority scheme employed, the expected averages of the wait time and queue length are computed. The expected average wait time gives an indication of the turn around time of a request. A small expected wait time is of necessity in many real time environments.

The expected average queue length is a function of the proportion of read to write requests and the priority between them. Although the proportion of read to write requests is fixed in this model, the maximum buffer size for each processor is assumed to be infinite. Further discussion on the queue length is presented in section 4.4.

For each simulation run of the model, the probability of acceptance and the expected wait time are reported for each memory configuration. Notice that if the total number of memory modules, N, is 2^n, there are n + 1 distinct memory configurations. For a given N, all the possible memory configurations were simulated. Two different sizes of memory systems, namely, N = 64 and N = 1024, were simulated for four processor

systems with p = 1, 2, 4, and 8, and five sets of module characteristics namely, $(a_r, c_r)/(a_w, c_w)$ = (1, 4)/(1, 6), (1,2)/(1, 4), (2, 2)/(2, 4), (2, 4)/(2, 6) and (4, 4)/(4, 6).

In each case, the simulation was performed for a fixed number of instruction cycles because of simulation costs. However, the simulation was exercised for a higher number of instruction cycles in selected cases and it was found that no significant changes of P_A (a, c, p) occurred after the fixed number of instruction cycles. Hence it was assumed that the results obtained for P_A (a, c, p) represent a steady state value for most practical purposes.

4.2 Nonbuffered Request Processor System

In the nonbuffered request processor system, when a request is rejected, its process is blocked for $c_r + 3$ segment time units. The rejected request is resubmitted one instruction cycle later with the same address. This procedure of resubmission of a rejected request is repeated until the request is accepted whereupon its process is unblocked. However, when the process is unblocked, it cannot issue a new request until the instruction cycle of the accepted request is completed.

Figure 4.2.1 shows the model of the nonbuffered request processor system with p = 1. When a request is accepted or rejected, its state is stored in the first segment of the process state unit which is shifted to the right every STU. In addition to the address of the request, its process number and function, the state of a request includes a status flag which indicates whether the request was accepted or

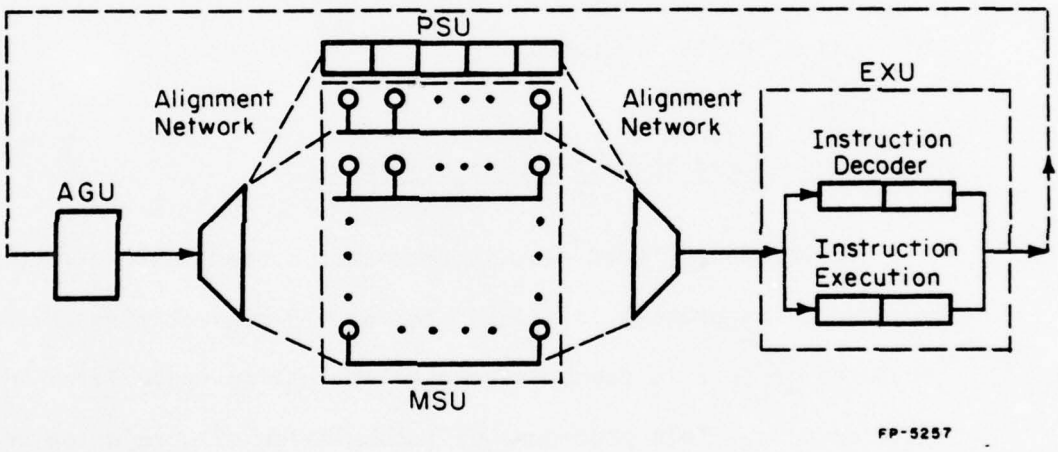


Figure 4.2.1 Nonbuffered request processor system for p = 1

rejected. For a read request, the request state is transmitted to the EXU segments after traversing the PSU segments for c_r STUs. If the status flag of the request indicates a rejection, the instruction is not processed, otherwise it is processed. The state of the write request also traverses the PSU segments for c_r STUs. However, since a write request requires $c_r + 2$ STUs for its servicing in the memory module, the process state of the write request is introduced into the EXU for 2 STUs but the write request requires no processing there.

In effect, each AGU and EXU contains registers for storing the states of active processes associated with a processor. The PSU is associated in time with the memory system unit (MSU).

This is not the only method of handling rejected requests between resubmissions. Instead of the recycling technique discussed above, a buffer could be used to store a rejected request for $c_r + 3$ STUs and retry the request at the beginning of its processes next instruction cycle. However, the control problems associated with this intermediate buffering technique are rather more involved than the recycling technique.

4.3 Buffered Request Processor System

In the buffered request processor system, there is one buffer for each of the p processors, each of which corresponds to s processes. All the requests issued by the AGU of each processor are queued up in that processor's buffer. Since there are s processes active in each processor, the queue for a processor will contain all the requests from

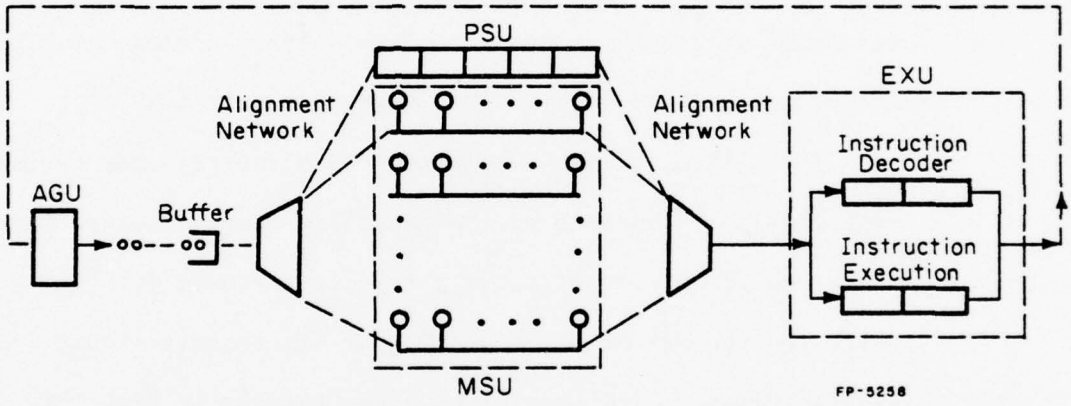


Figure 4.3.1 Buffered request processor system for p = 1

the s processes assigned to that processor. Each request issued by a processor joins the end of the queue for that processor, hence they are buffered in the order of their arrival. Figure 4.3.1 illustrates the BRP system for p = 1.

Each segment time unit, the buffered requests in each queue are simultaneously scanned in the order of their arrival. In each buffer, the first acceptable request, that is, the first request whose address corresponds to an idle line and module is selected for service. This queueing discipline produces at most p acceptable requests which then have to undergo the priority or multiple access line collision test to determine which of the acceptable requests to accept into the memory.

A process with a buffered read request is blocked until the request is selected, whereupon it is immediately unblocked. However, as pointed out in the NRP system, the process which has just been unblocked cannot issue a new memory request until the service of the previous request has been terminated. A process with a buffered read request is blocked because subsequent instructions or STUs of that process may depend on the data read by the buffered request. Therefore there is a dependency and further execution of that process cannot proceed until the buffered read request has been serviced.

Unlike the NRP system, a buffered write request does not cause its associated process to be blocked. A write request does not require postprocessing. Furthermore, the only dependency which may arise from a write request is that an instruction from the same process (assuming independent processes) which follows the write request may cause a reference to the same memory location as the write request.

Recall again that exactly one request is issued by each processor every segment time unit. However, at most one request per processor is selected for service every STU. Hence every STU, at most one serviced request is directed to its associated processor. Thus no collisions occur in the processor.

Since the requests in a buffer are scanned in the order of their arrival, a request may be selected for service before its predecessor, provided the request does not reference the same memory location and the preceding request was rejected. Hence, within a process, a request may be serviced before its predecessor if the predecessor is a write request. Notice that since a write request does not cause its associated process to be blocked, it permits a new request arrival from that process. Therefore, execution precedence within a process may not be preserved. However, it can be shown that although execution precedence within a process may not be preserved, the computational precedence maintained. Furthermore, the computational determinacy of a process is in question only when execution precedence is not maintained within the process. However, execution precedence is required between two requests in a buffer only if the requests refer to the same memory location and hence the same line and module. In such a case, the two requests must be serviced in the order of their arrival.

4.4 Discussion

For the nonbuffered case, the probability of acceptance, P_A (a, c, p) obtained from the simulation model is within about 6% of the corresponding

analytic results, hence, the graphical plots appear identical. Table 4.4.1 shows the analytic and experimental results in juxtaposition for (a, c) = (2, 4 2/3) and N = 64. Similarly, table 4.4.2 shows the analytic and experimental results for (a, c) = (2, 4 2/3) and N = 1024. From the tables, it can be seen that the analytic and experimental results are similar, especially for large values of N.

In general, the experimental results are less than their corresponding analytic results unless the difference is less than the error in our evaluation. In addition, as p increases, the difference between the experimental and analytic results becomes more apparent. For memory configurations (ℓ, m) such that $1 \leq \ell < ap$, the maximum service rate of requests, ℓ/a, is less than the request arrival rate, p, per STU. Hence there is excessive rejection of requests. In the experimental model for the NRP system, the rejected requests are resubmitted with the same module address one instruction cycle later. Hence the address distribution is not uniform and there is a tendency to reference lines and modules that cause the rejections more frequently without success. Therefore, the probability of rejection is higher for the experimental model.

It was pointed out earlier that the buffered case has not been modeled analytically because of the inherent complexity, hence, we do not have analytic results for the BRP system. However, the results of the experimental model for the BRP system will be tested by a comparison with the nonbuffered case. This will be discussed in more detail in the next chapter.

TABLE 4.4.1

P_A (a, c, p) for (a, c) = (2, 2 2/3) and N = 64

Memory Configuration (ℓ, m)	P_A (a, c, p), p = 2		P_A (a, c, p), p = 4	
	Analytic	Experimental	Analytic	Experimental
(1, 64)	0.2482	0.2501	0.1241	0.1250
(2, 32)	0.4244	0.4058	0.2393	0.2204
(4, 16)	0.6007	0.5745	0.3990	0.3764
(8, 8)	0.7474	0.7438	0.5712	0.5434
(16, 4)	0.8487	0.8431	0.7190	0.6996
(32, 2)	0.9097	0.9044	0.8231	0.8077
(64, 1)	0.9434	0.9316	0.8866	0.8774

TABLE 4.4.2

P_A (a, c, p) for (a, c) = (2, 4 2/3) and N = 1024

Memory Configuration (ℓ, m)	P_A (a, c, p), p = 2		P_A (a, c, p), p = 4	
	Analytic	Experimental	Analytic	Experimental
(1, 1024)	0.2499	0.2501	0.1249	0.1250
(2, 512)	0.4280	0.4028	0.2416	0.2228
(4, 256)	0.6072	0.5931	0.4047	0.3667
(8, 128)	0.7568	0.7421	0.5822	0.5506
(16, 64)	0.8604	0.8578	0.7360	0.7315
(32, 32)	0.9230	0.9233	0.8451	0.8414
(64, 16)	0.9576	0.9551	0.9120	0.9149
(128, 8)	0.9759	0.9731	0.9493	0.9510
(256, 4)	0.9853	0.9853	0.9691	0.9694
(512, 2)	0.9900	0.9902	0.9793	0.9784
(1024, 1)	0.9924	0.9935	0.9845	0.9847

Since there is excessive rejection of requests for memory configurations (ℓ, m) for $1 \leq \ell < ap$, some rejected requests may wait in the system unserviced indefinitely. Furthermore, $\varepsilon(W)$ and $\varepsilon(Q)$ may not reach steady state values in the simulation time period when ℓ is near ap.

$\varepsilon(W)$ and $\varepsilon(Q)$ are evaluated as follows. Let $\varepsilon_j(W)$ be the expected wait time of requests from processor j. Given that t_{ij} = time request (i, j) is accepted - time request (i, j) is issued,

$$\varepsilon_j(W) = \frac{1}{R_j} \sum_{i=1}^{R_j} t_{ij} ,$$

where (i, j) is the ith request from the jth processor and R_j is the total number of requests issued by processor j. Hence the expected average wait time is

$$\varepsilon(W) = \frac{1}{p} \sum_{j=1}^{p} \varepsilon_j(W),$$

where p is the number of processors. For some requests, t_{ij} cannot be evaluated since the corresponding requests are still in the system unaccepted by the end of the simulation.

In order to evaluate $\varepsilon(Q)$, let $q_j(t)$ represent the current queue length for processor j. The expected queue length for processor j is

$$\varepsilon_j(Q) = \frac{1}{T} \sum_{t=0}^{T} q_j(t) ,$$

where T is the total simulation time. Then the expected average queue length is

$$\varepsilon(Q) = \frac{1}{p} \sum_{j=1}^{p} \varepsilon_j(Q) .$$

In actual computation, each time an entry is inserted in or removed from the queue, the product of the old length of the queue and the interval of time over which this length existed is accumulated. If $\ell < ap$, the queue length will tend to grow to infinity with time. Hence, it is impossible to obtain a steady state evaluation of $\varepsilon(Q)$ in the finite simulation time period when $\ell < ap$.

5. ANALYSIS OF RESULTS

5.1 Introduction

In this chapter, the effects of varying the ℓ, N, a, c, p and τ parameters will be discussed with a view to understanding the design options available.

The primary performance indicator is the probability of acceptance, $P_A(a, c, p)$. $P_A(a, c, p)$ for a given memory configuration represents the system throughput as a fraction of the maximum possible. However, in order to compare and contrast systems with different module characteristics, (a, c), and processor order, p, the bandwidth is used as a performance indicator.

Definition 5.1.1 The bandwidth of an (ℓ, m) memory configuration accessed by a parallel-pipelined processor of order (s, p) is the expected number of accepted memory requests in one memory cycle and is given by $B(a, c, p) = pcP_A(a, c, p)$, where (a, c) are the module characteristics. □

Hence B(a, c, p) represents the system throughput per memory cycle. Notice that the bandwidth, B(a, c, p) is the effective parallelism in the memory.

When the read and write module characteristics differ, the effective module characteristics, namely, (a_e, c_e), will be used in place of (a, c). To illustrate the effect of the segment time unit, τ, on

performance, it is helpful to express the bandwidth as the expected number of accepted memory requests per second. In this case $B(a, c, p, \tau) = \frac{p}{\tau} P_A(a, c, p)$ requests/second.

A secondary performance indicator is the expected average wait time, $\varepsilon(W)$, which is the average time a request waits before it is serviced, assuming that all requests which arrive simultaneously have the same chance of being accepted. $\varepsilon(W)$ may be used in conjunction with $P_A(a, c, p)$ or $B(a, c, p)$ in the nonbuffered and buffered request processor systems to evaluate the effectiveness of various parameters. The expected average queue length, $\varepsilon(Q)$, is the average length of each queue, assuming that all p processors have identical priority. $\varepsilon(Q)$ is also used in evaluating the effectiveness of the buffered case.

The effects of parameter variation on performance are discussed primarily for the nonbuffered case. These effects can be easily extended to the buffered case. Since there is no exact solution for P_A in the general case, the lower bound is used occasionally to illustrate the effects of some of the parameters on the probability of acceptance, $P_A(a, c, p)$.

It is impractical to show the combined effects of all the parameters pictorially on a two-dimensional graph. A simplification, which is adopted here, studies the effect of each variable on performance, independently.

5.2 Effect of Number of Modules (N) on Performance

The total number of memory modules, N, can be interpreted in two ways. For a given memory size of M words, various memory configurations

are obtainable. By varying the size of the memory module, z, the total number of memory modules, N, can be varied. Alternatively, the total number of memory modules, N, represents the total size of the memory if the size of the memory module, z, is fixed.

In the discussion that follows, M is considered fixed and hence is irrelevant to the discussion that follows. The first interpretation of N is of primary interest. The determination of the absolute memory size, M, which usually involves page faulting and memory hierarchy considerations, is outside the scope of this research.

It is intuitively obvious that for a given number of lines, ℓ, and processor configuration, (s, p), an increase in N increases P_A. However, the relative increase cannot be obtained intuitively. In some situations, doubling N may not significantly increase P_A.

Figures 5.2.1, 5.2.2, and 5.2.3, which were obtained from the results of theorems 3.5.2 and 3.5.3, illustrate the effect of $N(\geq \ell)$ on P_A for $\ell = 4$, 16 and 64 respectively. In general, an increase in N increases the performance for given ℓ, a, c and p.

As N approaches infinity, the lower bound on P_A of theorem 3.6.2 approaches the upper bound on P_A of theorem 3.6.3 as shown below. Recall that $1 - P_1 = [1 - (1-\frac{1}{\ell})^p] \frac{\ell}{p}$ and the lower bound is given by

$$P_A(a, c, p) \geq \frac{1 - P_1}{1 + (1-P_1)\left[\frac{p(a-1)}{\ell} + \frac{p(c-a)}{N}\right]}$$

$$\lim_{N \to \infty} P_A(a, c, p) \leq \frac{1 - P_1}{1 + (1-P_1)\frac{p(a-1)}{\ell}}$$

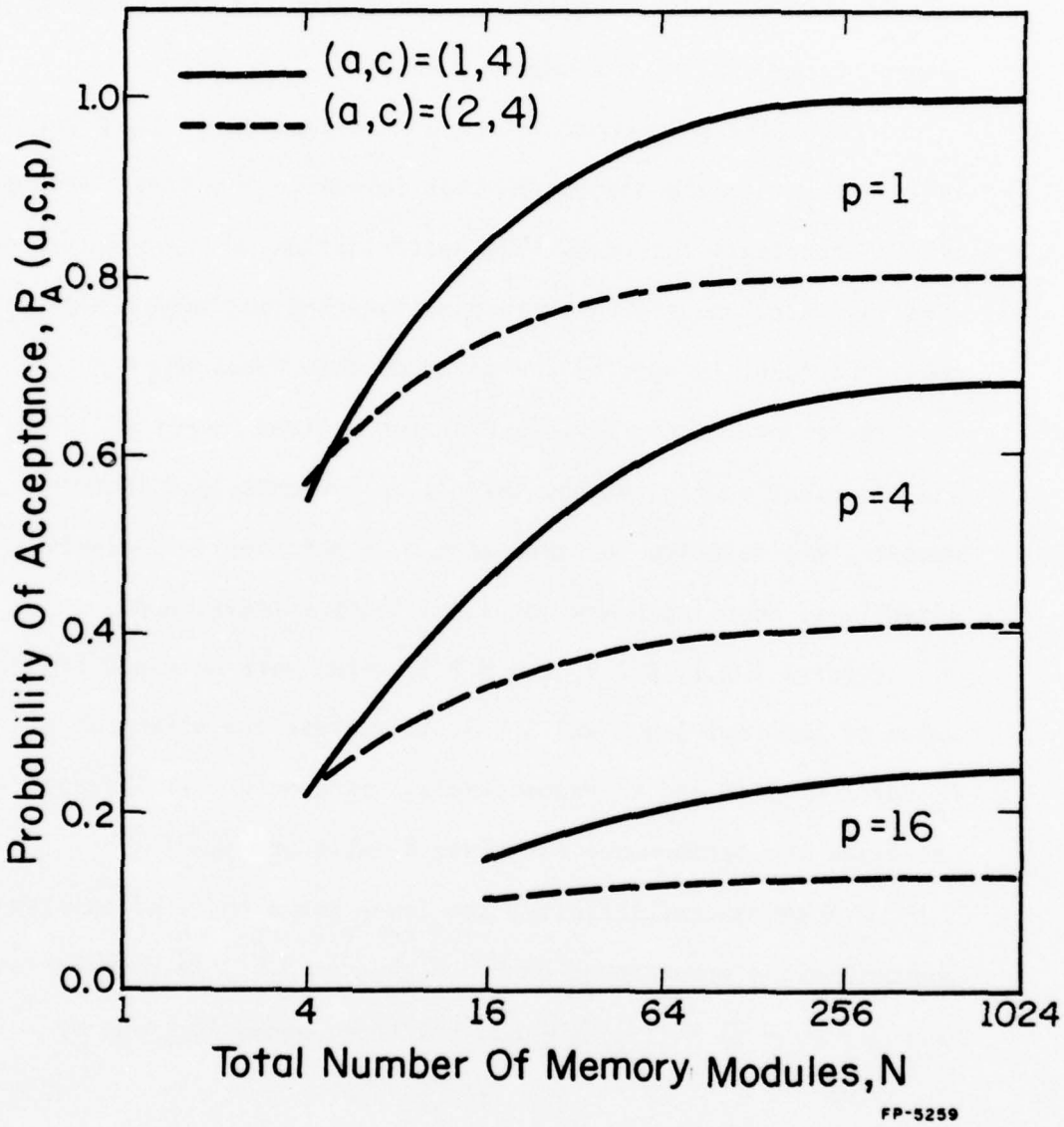


Figure 5.2.1 Effect of N on P_A for $\ell = 4$

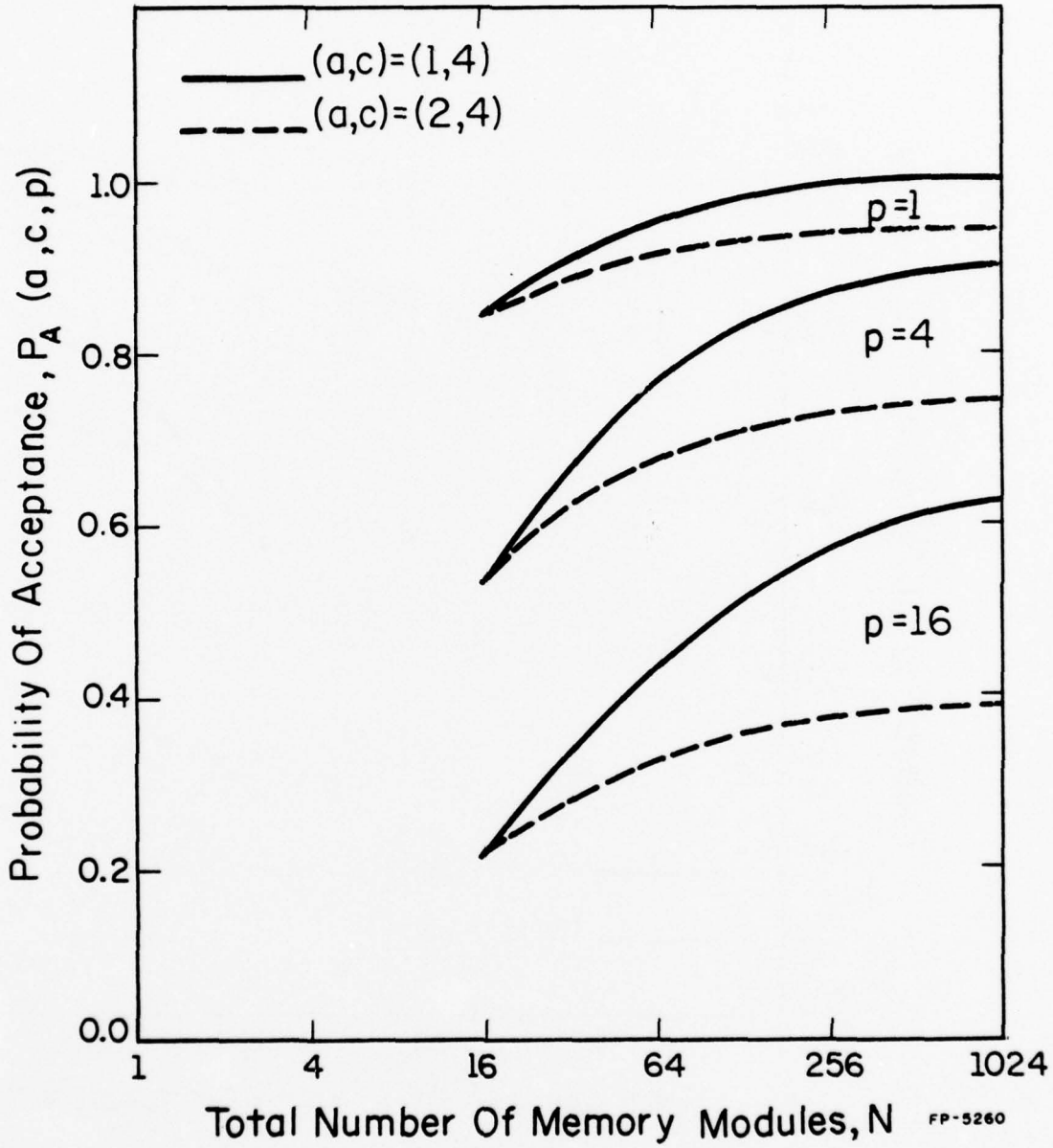


Figure 5.2.2 Effect of N on P_A for $\ell = 16$

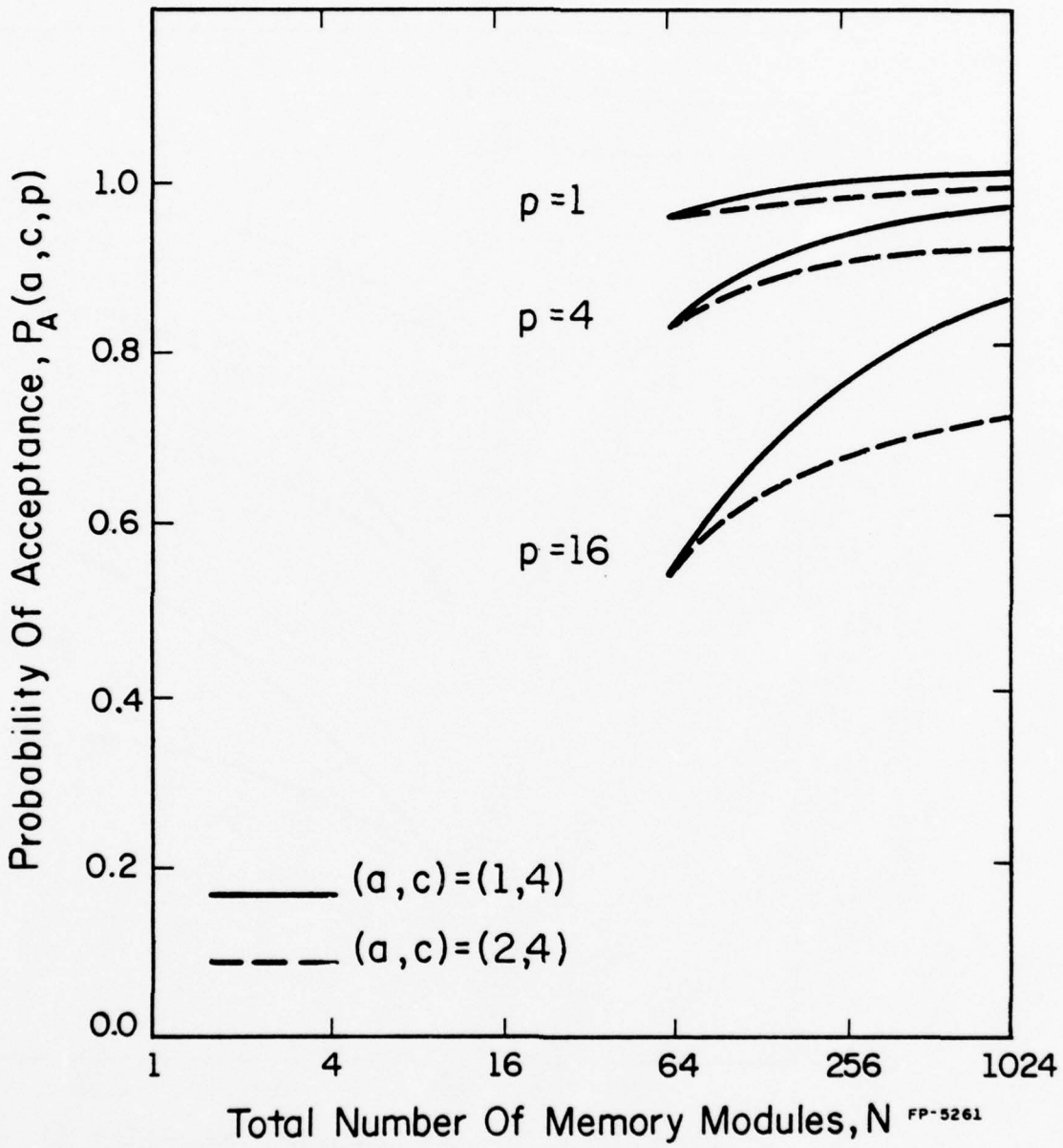


Figure 5.2.3 Effect of N on P_A for $\ell = 64$

But the upper bound of theorem 3.6.3 says $P_A(a, c, p) \leq \frac{1 - P_1}{1 + (1-P_1)\frac{p(a-1)}{\ell}}$. Thus,

$$\lim_{N \to \infty} P_A(a, c, p) = (1 - P_1)/[1 + (1 - P_1)\frac{p(a-1)}{\ell}].$$

From the bounds expressions above, it is seen that for very large N, the memory cycle, c, does not have a significant effect on P_A. Note that c is usually a small number in most systems, generally less than 10. Hence, for large N, P_A is limited by ℓ and p. The graphs also show that the address cycle, a, highly affects P_A for large N. As an illustration consider figure 5.2.1, which is for $\ell = 4$. Suppose that $p = 4$ and P_A is required to be 0.4. Using $(a, c) = (2, 4)$, N is required to be at least 256, whereas, if $(a, c) = (1, 4)$, N is required to be only 16.

For large ℓ and small p, and for small ℓ and large p, there is progressively less payoff (increase in P_A) to increasing N. However, for small ℓ and small p, and large ℓ and large p, there is some significant payoff to increasing N. Figures 5.2.1 and 5.2.3 illustrate these phenomena. It is obvious from a glance at the figures that the choice of N is critical in a certain range of memory configurations and processor systems. However, it is clear that the choice of N cannot be made independently from other parameters. This discussion will be exemplified by some case studies in section 5.8.

5.3 Effect of the Number of Lines on Performance

It is obvious from the collision considerations that an increase in ℓ reduces the line collisions and multiple access line collisions and hence increases the performance. Figures 5.3.1 and 5.3.2, which were

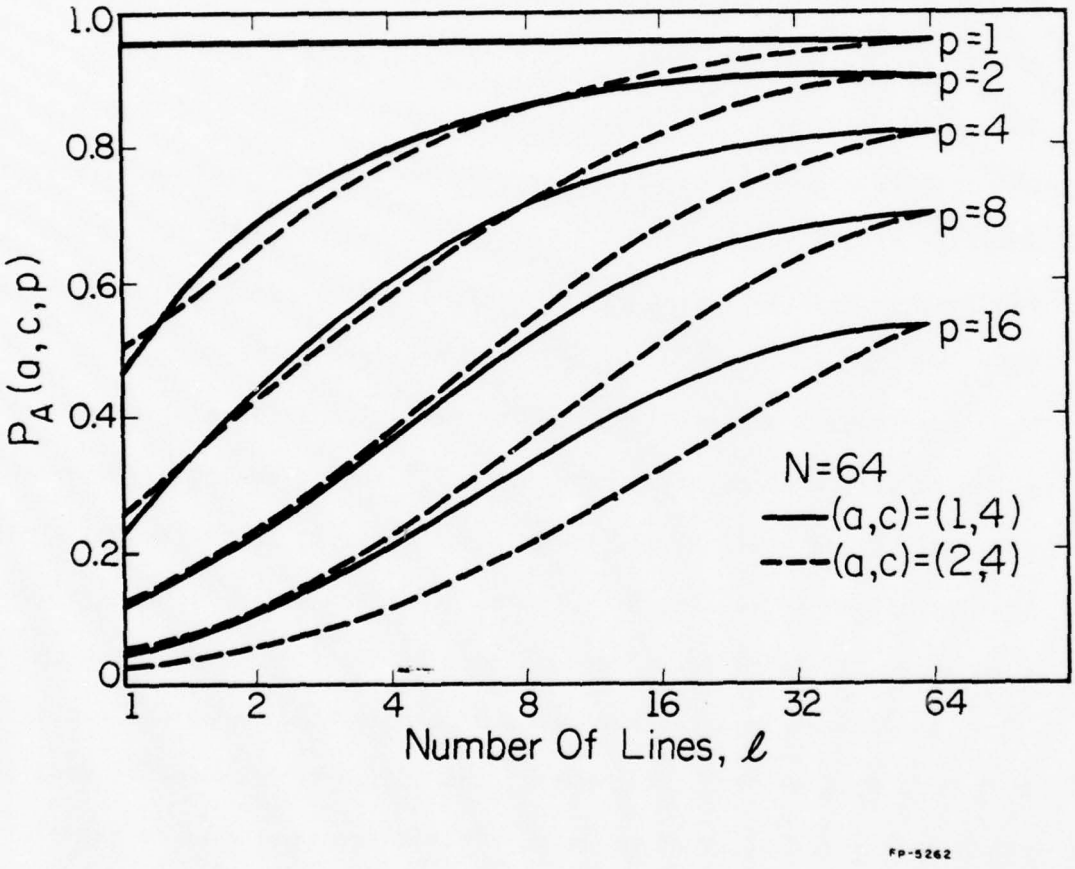


Figure 5.3.1 Effect of ℓ on P_A for N = 64

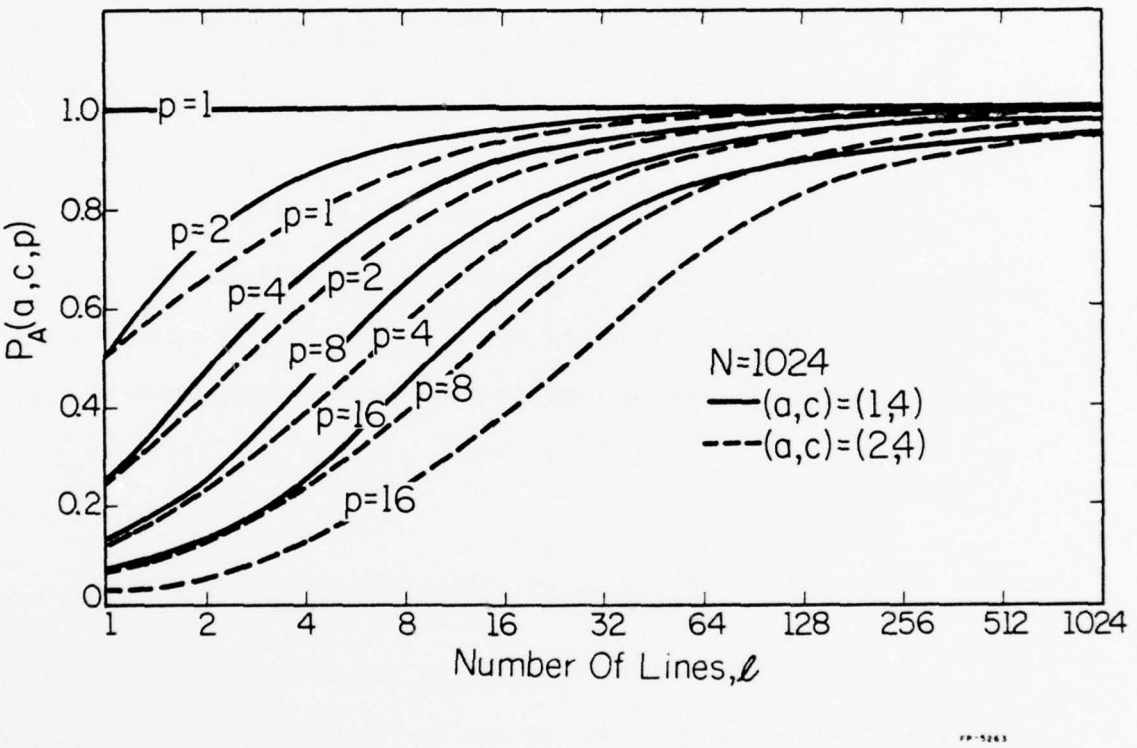


Figure 5.3.2 Effect of ℓ on P_A for N = 1024

obtained from the results of theorems 3.5.2 and 3.5.3, illustrate the effect of the number of lines on P_A for N = 64 and N = 1024 respectively.

For $\ell << p$, the lines are saturated. Hence in this region there is very little payoff in performance to doubling ℓ. It is obvious that configurations with $\ell < p$ result in poor and hence undesirable performance because of the excessive line saturation in this region. Furthermore, the line saturation worsens as a increases.

There is a point of inflection at $\ell = p$. For ℓ in the neighborhood of p and ap, there is a significant increase in P_A for an increase in ℓ. Hence this is the region in which the partial derivative of P_A with respect to ℓ is greatest. Beyond $\ell = ap$, P_A is close to 1. Hence, increase in ℓ increases P_A which asymptotically approaches the upper bound of theorem 3.6.3. Notice that in this case the asymptote is $(1 - P_1)/[1 + \frac{(1 - P_1)\ p(a-1)}{N}]$, where $1 - P_1 = [1 - (1 - \frac{1}{N})^P]\frac{N}{P}$.

5.4 Effect of Module Characteristics on Performance

It is obvious that an increase in the address cycle, a, increases the line collisions, whereas an increase in the memory cycle, c, increases the module collisions. Hence as a or c increases, the probability of acceptance decreases. The effects are, however, minimal when ℓ and N, respectively, are sufficiently large.

Recall that a reduction in ℓ increases the number of line collisions and multiple access line collisions. Hence as a increases for small ℓ, P_A is sharply reduced. As ℓ increases, the line collisions and multiple access line collisions are reduced and P_A is less sensitive to an increase

in a. This effect is illustrated in figure 5.4.1, where P_A is plotted against (a, 4) for a = 1, 2, 3 and 4 for various memory configurations. Figures 5.4.1 and 5.4.3 were obtained from results of theorems 3.5.2 and 3.5.3.

The effect of a can also be explained somewhat analytically by using the lower bound formula for P_A, which is

$$\frac{1 - P_1}{1 + (1 - P_1)\left[\frac{p(a-1)}{\ell} + \frac{p(c-a)}{N}\right]}$$

There are essentially three regions of ℓ in which the effects of a are highlighted. For small ℓ such that $\ell < p$ and $\ell \ll N$, the lines are saturated and $P_A \triangleq \frac{\ell}{ap}$. Hence P_A is inversely proportional to a. For ℓ nearly N, an increase in a increases the $\frac{p(a-1)}{\ell}$ term while decreasing the $\frac{p(c-a)}{N}$ term. The overall effect is a slight increase in the denominator of the lower bound expression with increase in a. Hence P_A decreases slightly with increase in a for ℓ nearly N. When $\ell = N$, theorem 3.6.1 can be applied. It was seen then that for $\ell = N$, P_A is independent of a. Hence P_A remains constant for $\ell = N$ as a increases.

In section 5.2, it was shown that P_A is insensitive to c for large N ($>>pc$). This insensitivity is illustrated in figure 5.4.2, which was obtained from results of theorem 3.5.2, for N = 256 and p = 4. It should be noted that c cannot however be arbitrarily large, since $c < s$, which is the degree of pipelining.

The combined effect of a simultaneous increase in a and c is illustrated in figure 5.4.3 for c/a = 2. It is also seen here that a is critical in determining P_A.

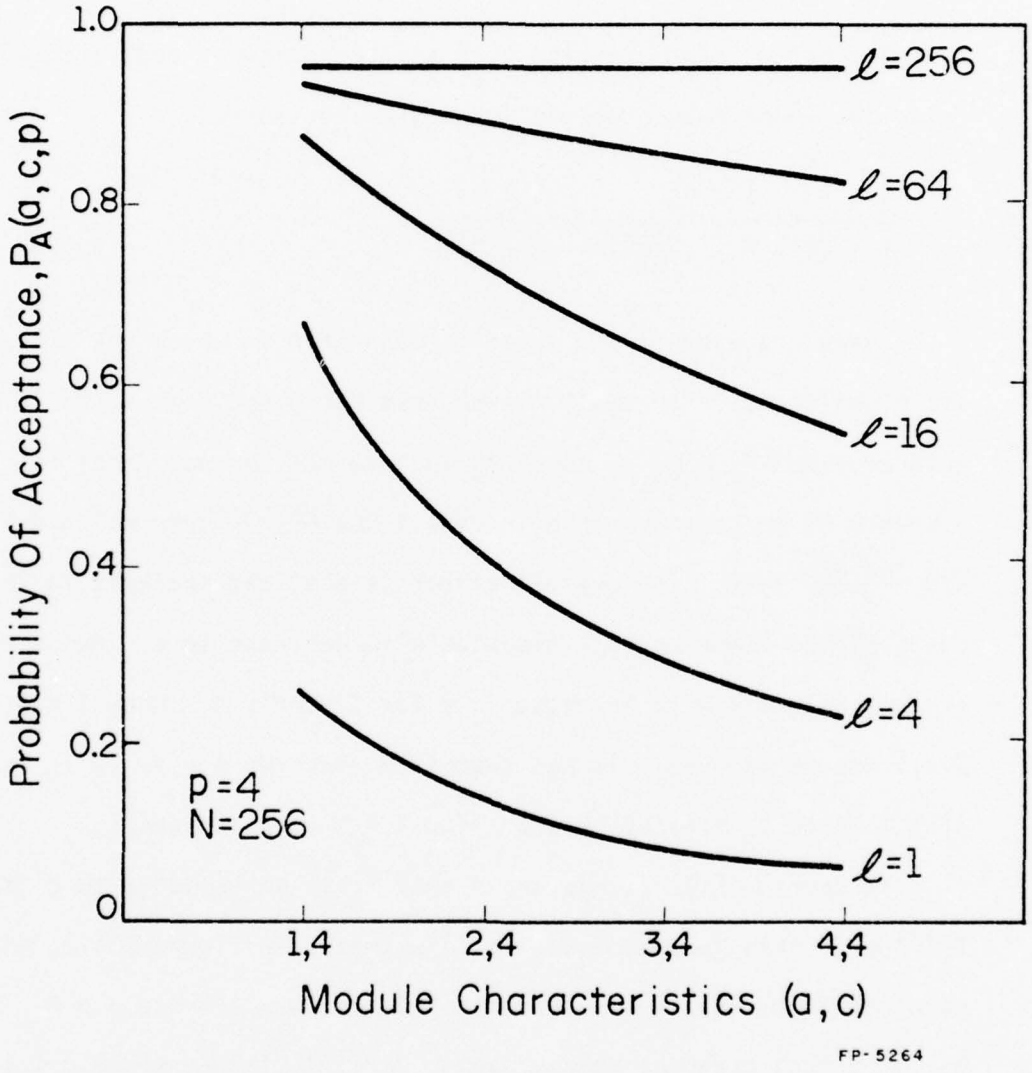


Figure 5.4.1 Effect of (a, c) on P_A for c = 4, p = 4, and N = 256

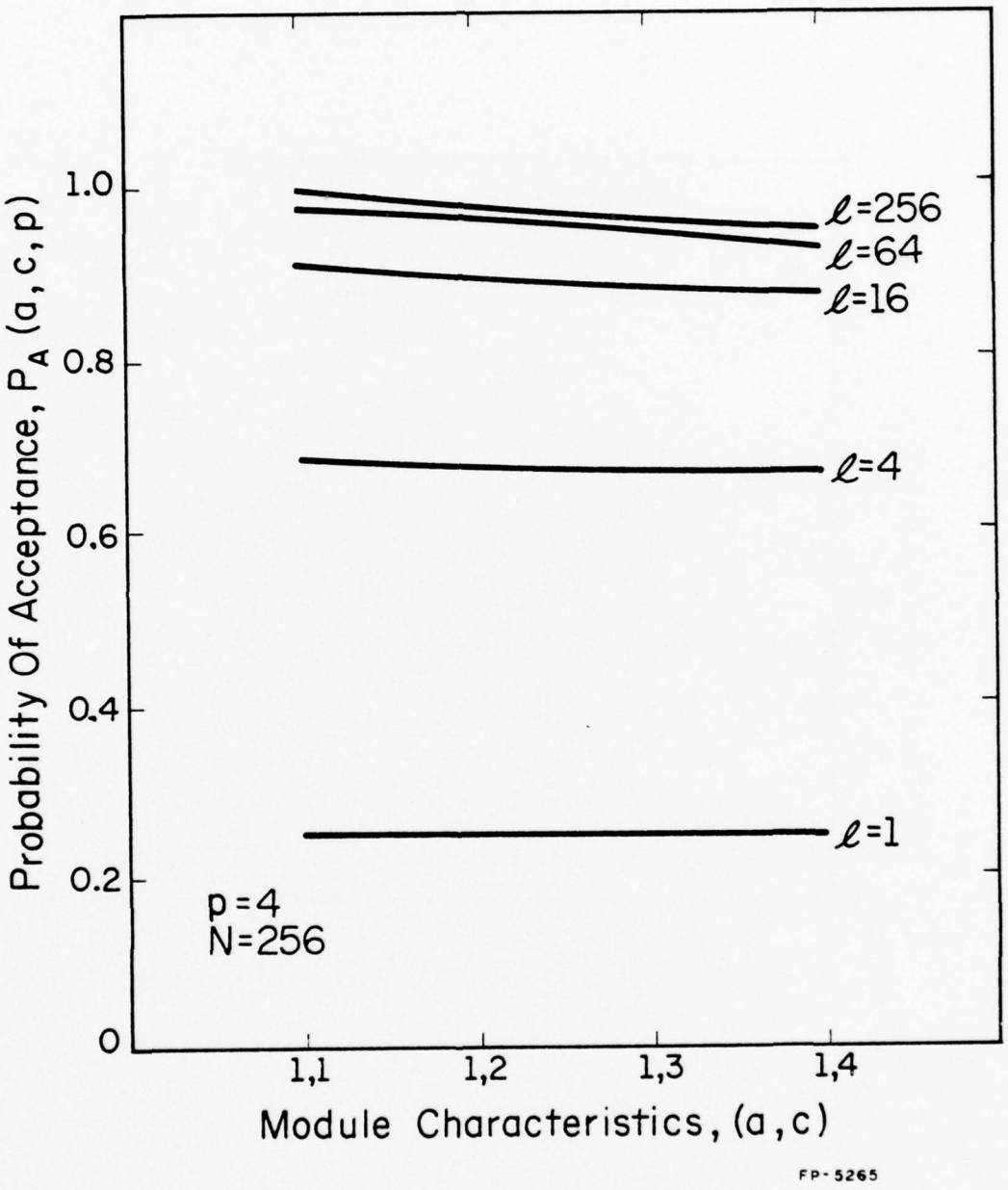


Figure 5.4.2 Effect of (a, c) on P_A for a = 1, p = 4, and N = 256

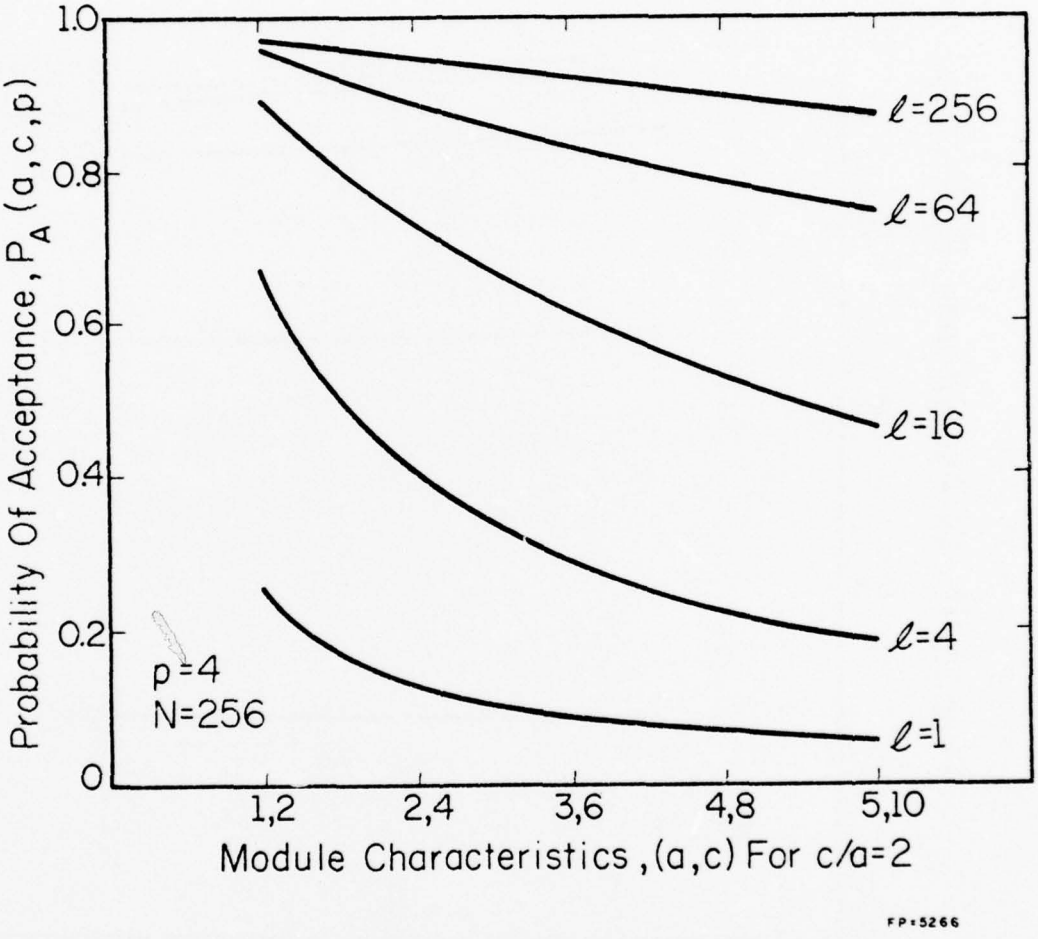


Figure 5.4.3 Effect of (a, c) on P_A for $\frac{c}{a} = 2$, p = 4, and N = 256

It appears that a reasonable conclusion is that in general, if N is high enough, variations in c have little effect on P_A for any configuration. If ℓ is large or ℓ is close to N, variation in a has little effect on P_A.

5.5 Effect of Processor Order on Performance

The choice of p is very critical in obtaining a reasonable performance. Once again, employing the lower bound formula for P_A, the sensitivity of P_A to p can be evaluated for some class of p. For large p such that $p \gg \ell$, $(1 - \frac{1}{\ell})^p \triangleq 0$. Hence, $P_A \triangleq \frac{\ell/p}{a + \frac{(c-a)\ell}{N}} \triangleq \frac{\ell}{ap}$, for large N. In this region, P_A is small. Furthermore, increasing p without limit decreases P_A asymptotically to zero. This effect is illustrated in figure 5.5.1. However, for small p such that $p \ll \ell$, $(1 - \frac{1}{\ell})^p \triangleq 1 - \frac{p}{\ell}$. Hence, $P_A \triangleq \frac{1}{1 + \frac{p(a-1)}{\ell} + \frac{p(c-a)}{N}}$. In this region, P_A is close to 1. An increase in p does not affect P_A significantly as long as $p \ll \ell$ and $N \geq \ell$. For example, in figure 5.5.1, if (a, c) = (2, 4), ℓ = 64 and p = 2, an increase in p by 100% (i.e., to 4) decreases P_A by less than 6%. However, for smaller ℓ or larger p, P_A becomes more sensitive as is evidenced for ℓ = 16 or p = 16 to 32.

The total number of memory modules, N, influences the effect of p on P_A. An increase in N shifts the curves up, that is, toward P_A = 1 and flattens them, making them less sensitive to p. Similarly, the sensitivity of P_A to p is accentuated by a decrease in N.

However, if bandwidth is the performance criterion, then as illustrated in figure 5.5.2, an increase in p increases the bandwidth generally.

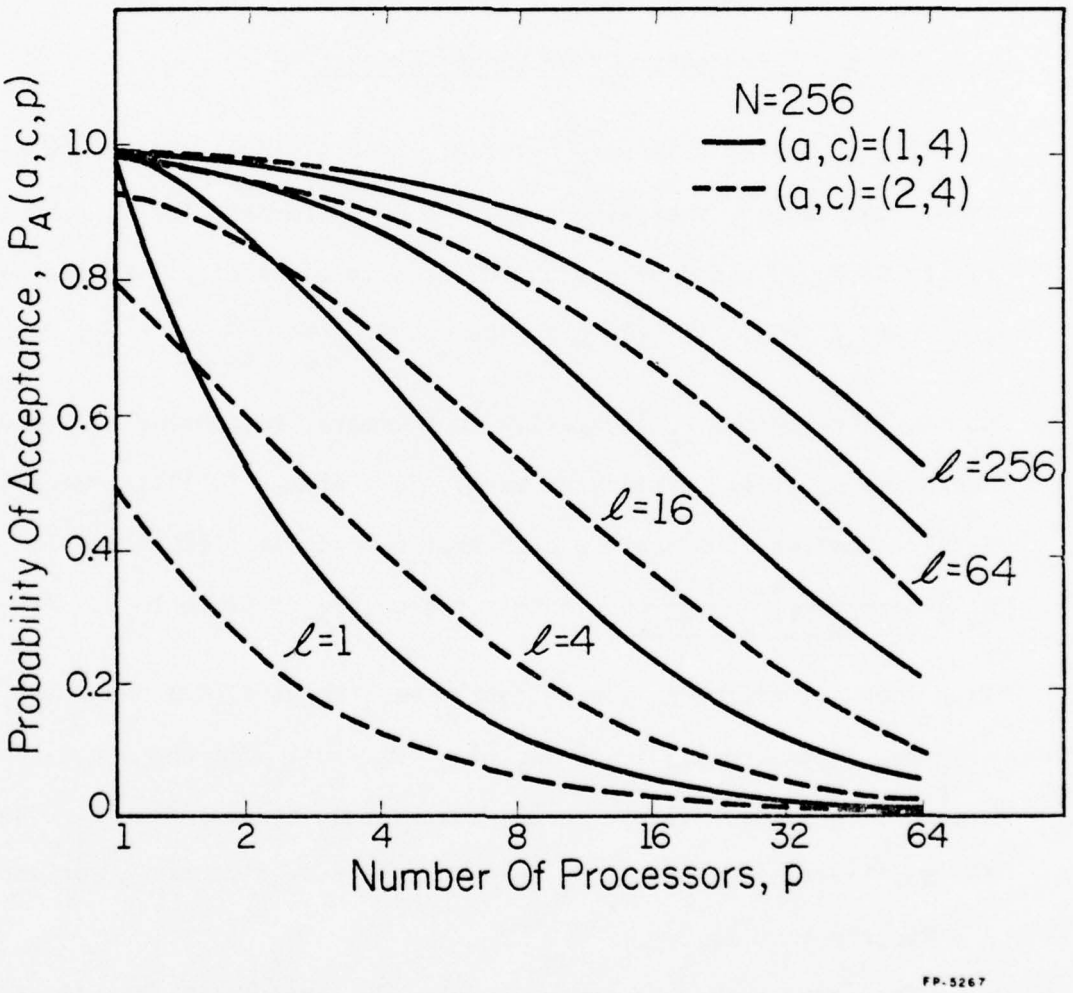


Figure 5.5.1 Effect of p on P_A for N = 256

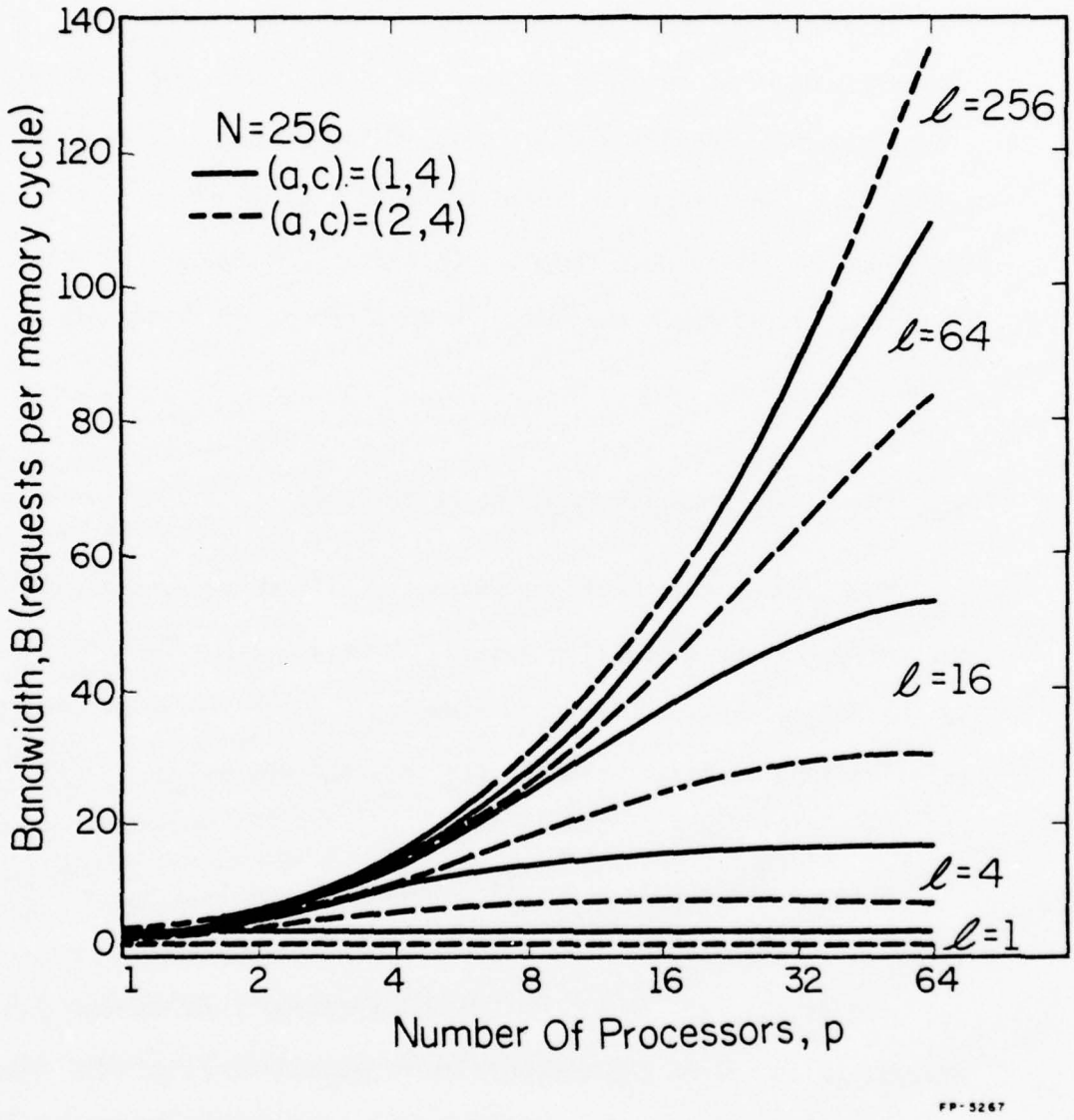


Figure 5.5.2 Effect of p on B for N = 256

The leveling off of the bandwidth curves for $p > \ell$ also illustrates the saturation of the lines. For $p \ll \ell$, P_A is close to 1 and an increase in p increases the bandwidth almost linearly.

In summary, for $\ell >> p$ and very large N, P_A is close to 1 and is not very sensitive to small variations in p. Hence the bandwidth increases almost linearly with p. As p/ℓ increases to 1, P_A becomes very sensitive to variations in p and there is a point of inflection in the bandwidth curves in this region. Once $p > \ell$, P_A decays asymptotically to zero as p increases further. In this region the bandwidth curves flatten.

5.6 Effect of Processor Speed on Performance

The segment time unit, τ, was not really considered explicitly in the analytical or simulation models. However, its effect can be explained in the following ways with some examples. Recall that the absolute and relative module characteristics are (a_o, c_o) and (a, c) respectively, where $a = \left\lceil \frac{a_o}{\tau} \right\rceil$ and $c = \left\lceil \frac{c_o}{\tau} \right\rceil$.

Suppose the absolute module characteristics (a_o, c_o) are given as (200, 400) in nanoseconds. If τ = 200, 100 and 50 nanoseconds, then (a, c) = (1, 2), (2, 4) and (4, 8) respectively. In section 5.4, the effects of (a, c) on the probability of acceptance, P_A, were discussed. Recall that for large N and small values of ℓ, a simultaneous increase in a and c decreases P_A exponentially. Thus a decrease in τ causes a decrease in P_A. In this case however, P_A is not an appropriate performance indicator of the effect of τ since decreasing τ also increases the

rate of requests to the memory. Instead, the absolute bandwidth, $B(a, c, p, \tau)$, is adopted as a measure of performance. $B(a, c, p, \tau) = \frac{p}{\tau} P_A(a, c, p)$.

Three examples are developed below to assess the effects of decreasing τ. These correspond, respectively, to a constant request rate assumption, a higher request rate assumption, and a uniformly faster processor/memory system assumption.

Suppose, for a first example, the number of processes which request memory within one memory cycle is fixed at I_o as τ varies. Then $I_o = pc$. Therefore doubling the speed of the processor (halving τ) doubles c and, in effect, halves p in order to keep I_o constant.

Hence $B(a, c, p, \tau)$ can be rewritten as $\frac{I_o}{c\tau} P_A(a, c, p) \approx \frac{I_o}{c_o} P_A(a, c, p)$ where p is proportional to τ and c is inversely proportional to τ. Figure 5.6.1, which is obtained from results of theorems 3.5.2 and 3.5.3, illustrates the example given above for $I_o = 8$, $(a_o, c_o) = (200, 400)$ ns, N = 256 and p goes from 1 to 4 as τ goes from 50 to 200. Observe that an increase in the speed of the processor (decrease in τ), reduces the bandwidth especially for configurations with small ℓ. For large values of ℓ, the reduction is less noticeable. Recall that a reduction in p increases the probability of acceptance. The reduction in the bandwidth is due to a combination of the contrary effects of reducing p while increasing (a, c) simultaneously as the speed is increased (τ is decreased).

The above example illustrates one important factor. For a constant I_o, a decrease in processor speed increases p and decreases c simultaneously. For the constant I_o assumption, doubling τ halves s. Hence sp is constant as τ changes. Under the constant I_o restriction, it can be

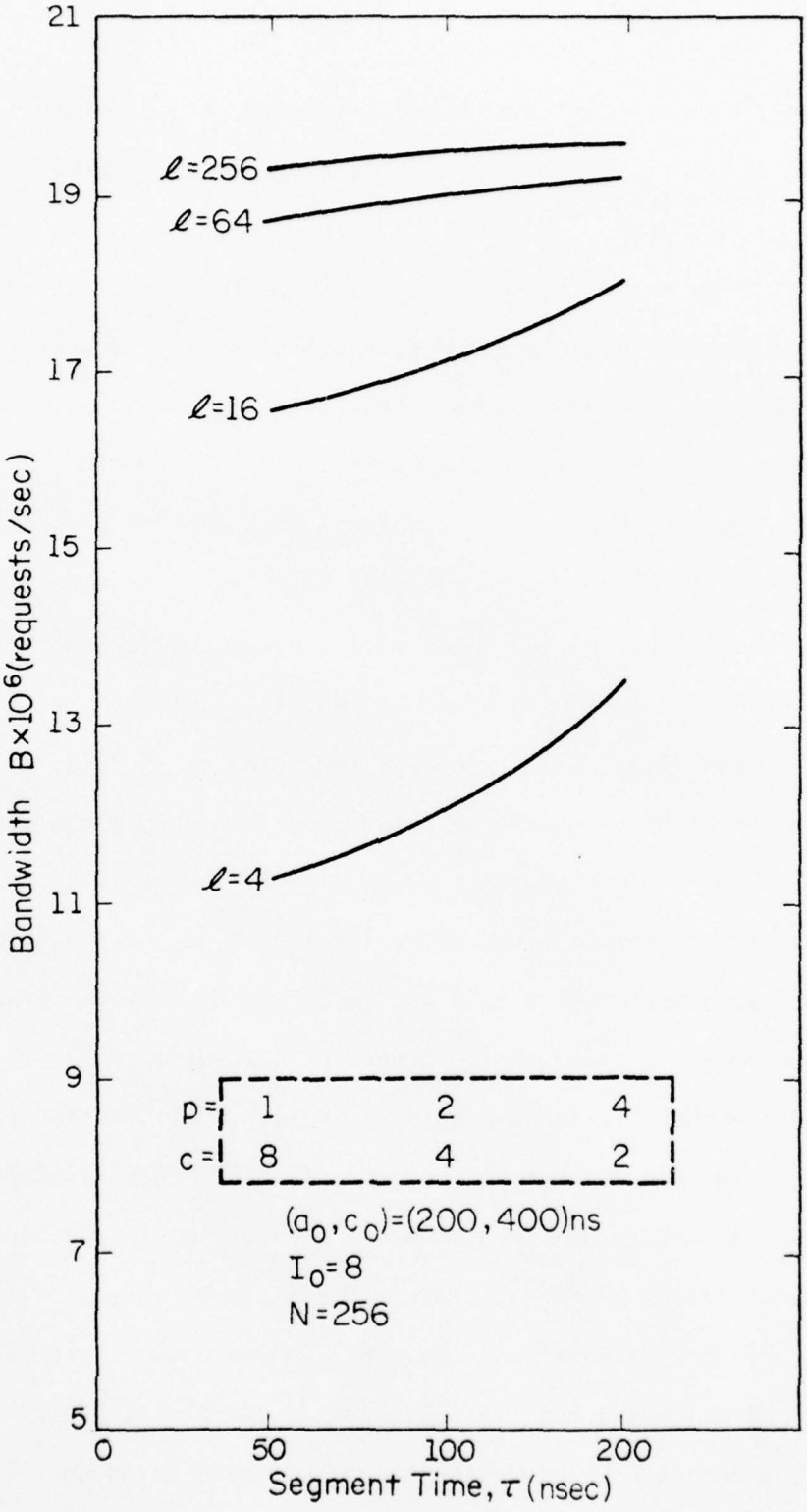


Figure 5.6.1 Effect of Processor Speed on Bandwidth for a Constant Request Rate

FP-5269

deduced from figure 5.6.1 that increasing τ increases the bandwidth. Hence for a constant sp, bandwidth is maximized by maximizing p. In the above example, a change in τ corresponds to physical changes in the processor design.

Consider a second example in which l_o is not fixed. Assume that N, p and (a_o, c_o) are fixed but τ varies. Note that s will still vary inversely with τ and therefore may necessitate major changes in the processor system. Figure 5.6.2, which is obtained from results of theorems 3.5.2 and 3.5.3, illustrates the higher request rate example for N = 256, p = 4 and (a_o, c_o) = (200, 400) ns. In this case, an increase in the processor speed increases the degree of pipelining, s, and the rate of requesting memory. In general, an increase in the processor speed increases the bandwidth. For large ℓ, P_A approaches 1 and the bandwidth is inversely proportional to τ.

In a third example, it is assumed that l_o, N, s, p, a and c are fixed. Decreasing τ then simply corresponds to faster clocking of the processor. Furthermore, decreasing τ requires a proportional decrease in (a_o, c_o) so that (a, c) is fixed. For an example, let N = 256, p = 4, and (a, c) = (1,2). $P_A(a, c, p)$ is a constant for a given memory configuration as processor and memory speed changes. Figure 5.6.3, which is obtained from the results of theorem 3.5.2, shows the effect of the processor and memory speed on the bandwidth for various memory configurations. In all configurations, an increase in speed (decrease in τ) increases the bandwidth proportionally. Notice that the bandwidth relative to the memory cycle is constant for a given (ℓ, m) as the speed changes. In fact, doubling the speed doubles the absolute bandwidth.

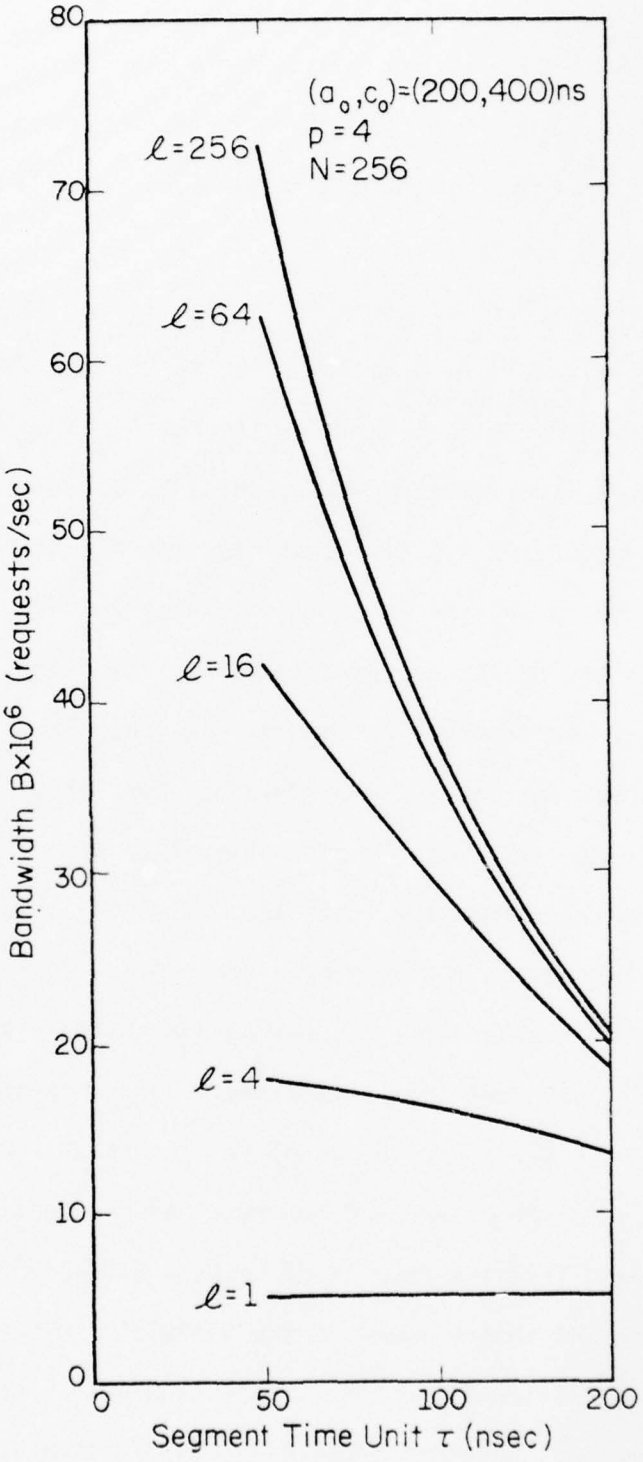


Figure 5.6.2 Effect of Processor Speed on Bandwidth for Varying Request Rate

FP-5270

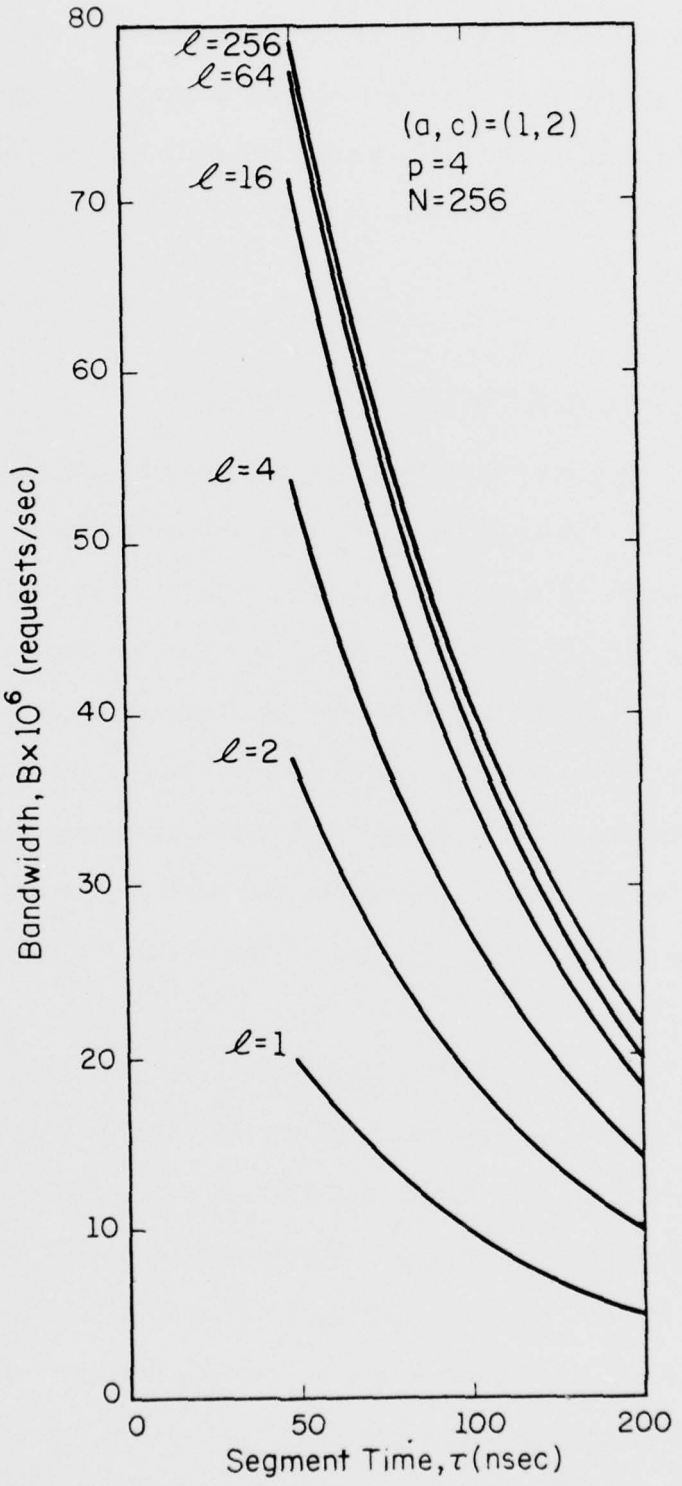


Figure 5.6.3 Effect of Processor and Memory Speed on Bandwidth Uniform Memory System FP-5271

This is expected since doubling the processor speed necessitates a new memory module whose absolute module characteristics are half the previous ones. In this example, a change in the processor speed will therefore require a change of the old memory system to a new one as explained above.

5.7 Effect of Buffering on Performance

For any configuration, the buffered scheme produces as good or better probability of acceptance, P_A than the nonbuffered scheme. All illustrations in this section are obtained from simulation results. Examples are illustrated in figures 5.7.1 and 5.7.2. Recall that a processor's buffer is scanned every STU for an acceptable request. This process is more likely to produce an acceptable request than the recycling technique for the NPR system. Hence for the BPR system, a rejected request remains in the queue which is rescanned the next STU, thus, giving the rejected request another trial if none of the preceding requests is an acceptable request.

Figures 5.7.3 and 5.7.4 give a very good indication of the effect of buffering on the bandwidth for N = 64 and N = 1024 respectively, for $(a_e, c_e) = (2, 4.67)$. Recall that (a_e, c_e) are the effective module characteristics which take into account the read and write module characteristics and request distribution. A point of inflection occurs at $\ell = p$. For $\ell < p$, the lines are saturated, resulting in excessive blocking of processes. Again, a configuration with $\ell < p$ is not desirable.

In general, the bound of theorem 3.6.4 applies to the buffered case as well as the nonbuffered case. An example of this bound is shown in

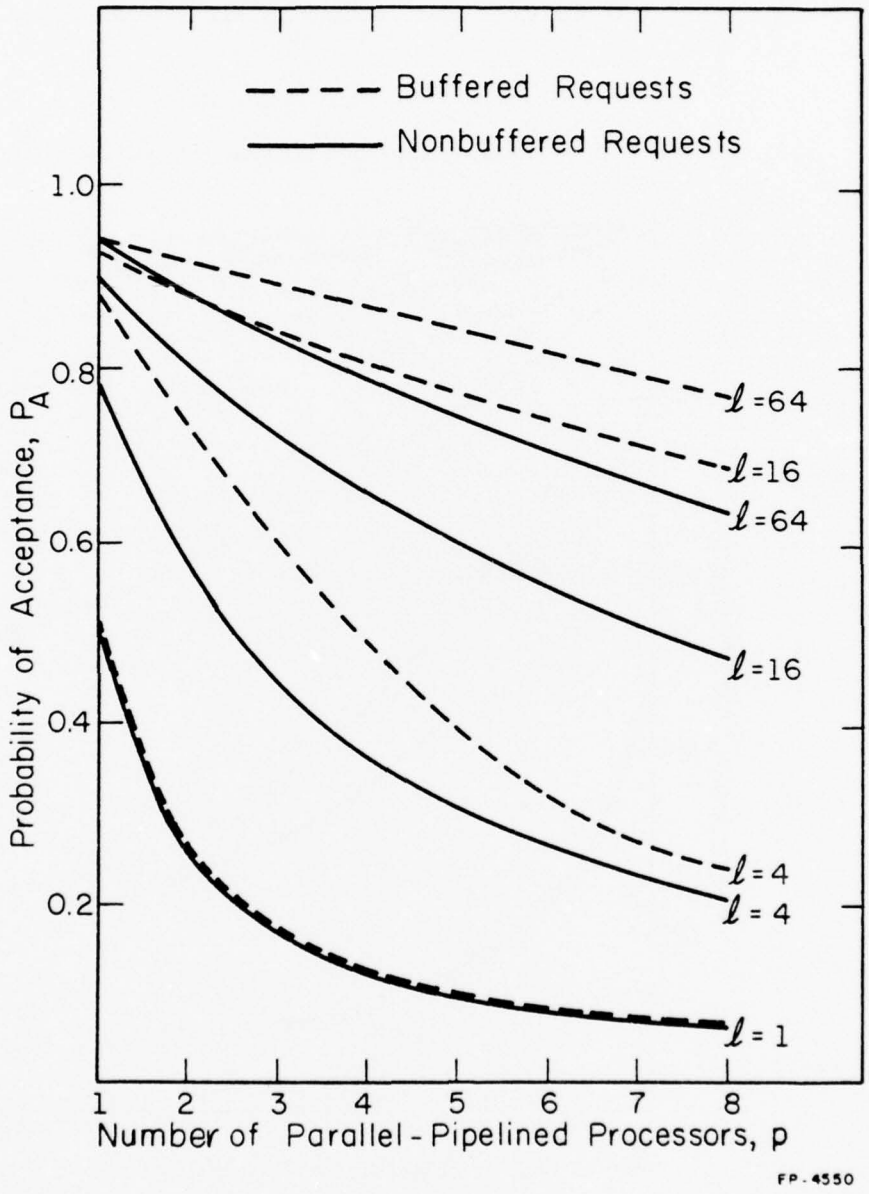


Figure 5.7.1 Probability of Acceptance Versus Number of Processors for Some Configurations, where N = 64 and (a_e, c_e) = (2, 4.67)

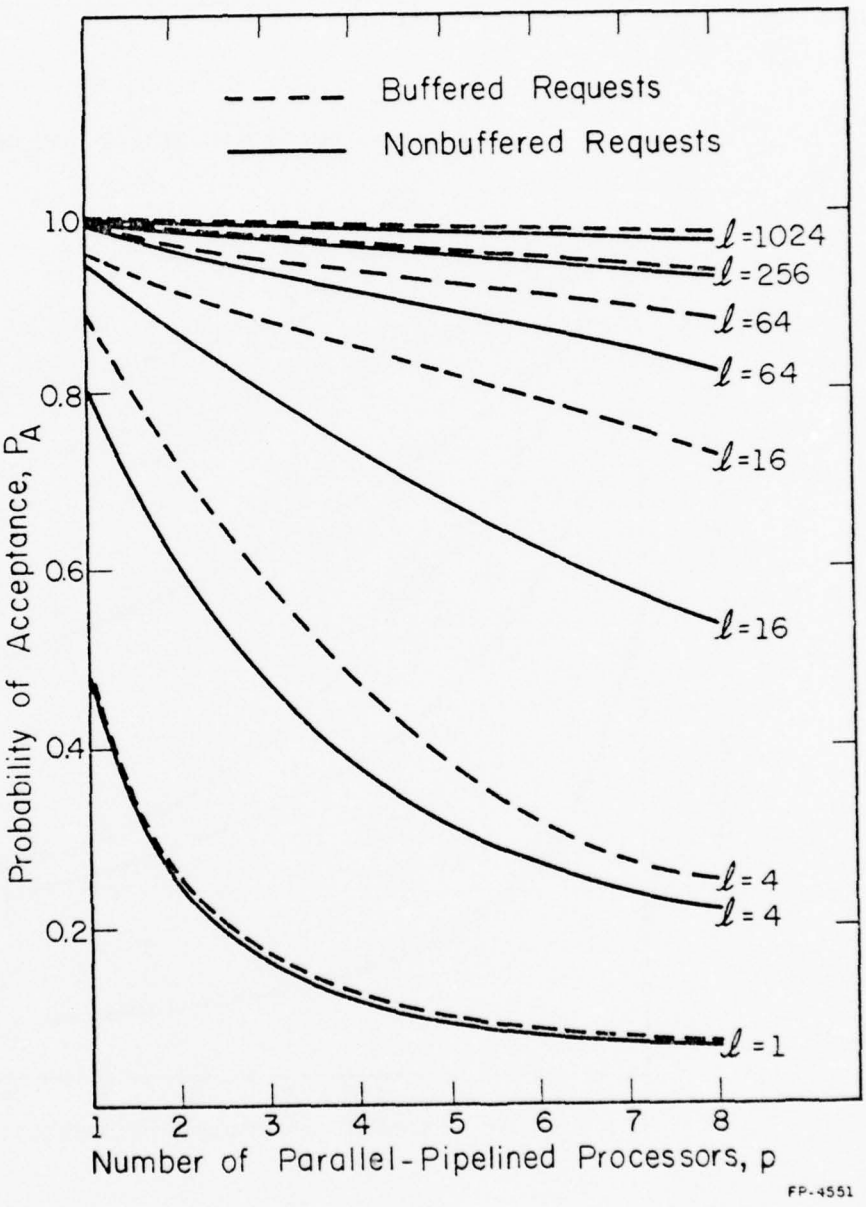


Figure 5.7.2 Probability of Acceptance Versus Number of Processors for Some Configurations, where N = 1024 and (a_e, c_e) = (2, 4.67)

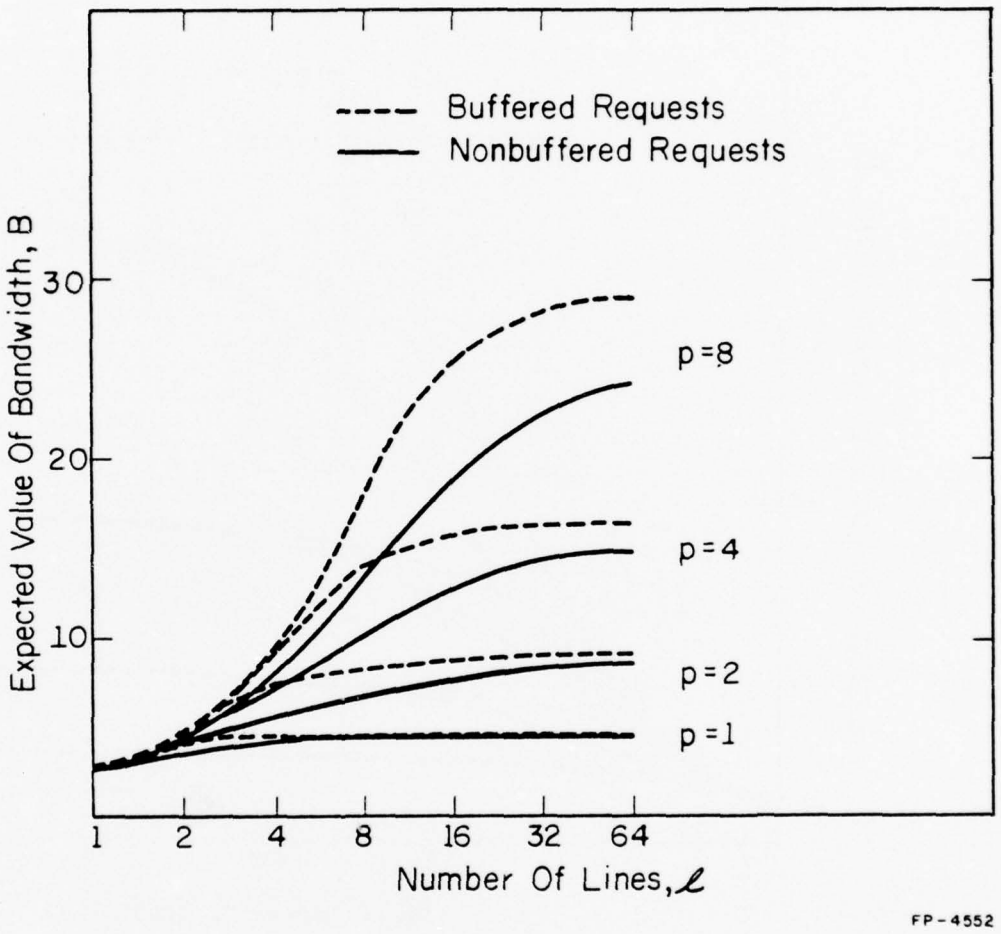


Figure 5.7.3 Bandwidth Versus Memory Configuration for Various Values of p (number of processors), where N = 64 and $(a_e, c_e) = (2, 4.67)$

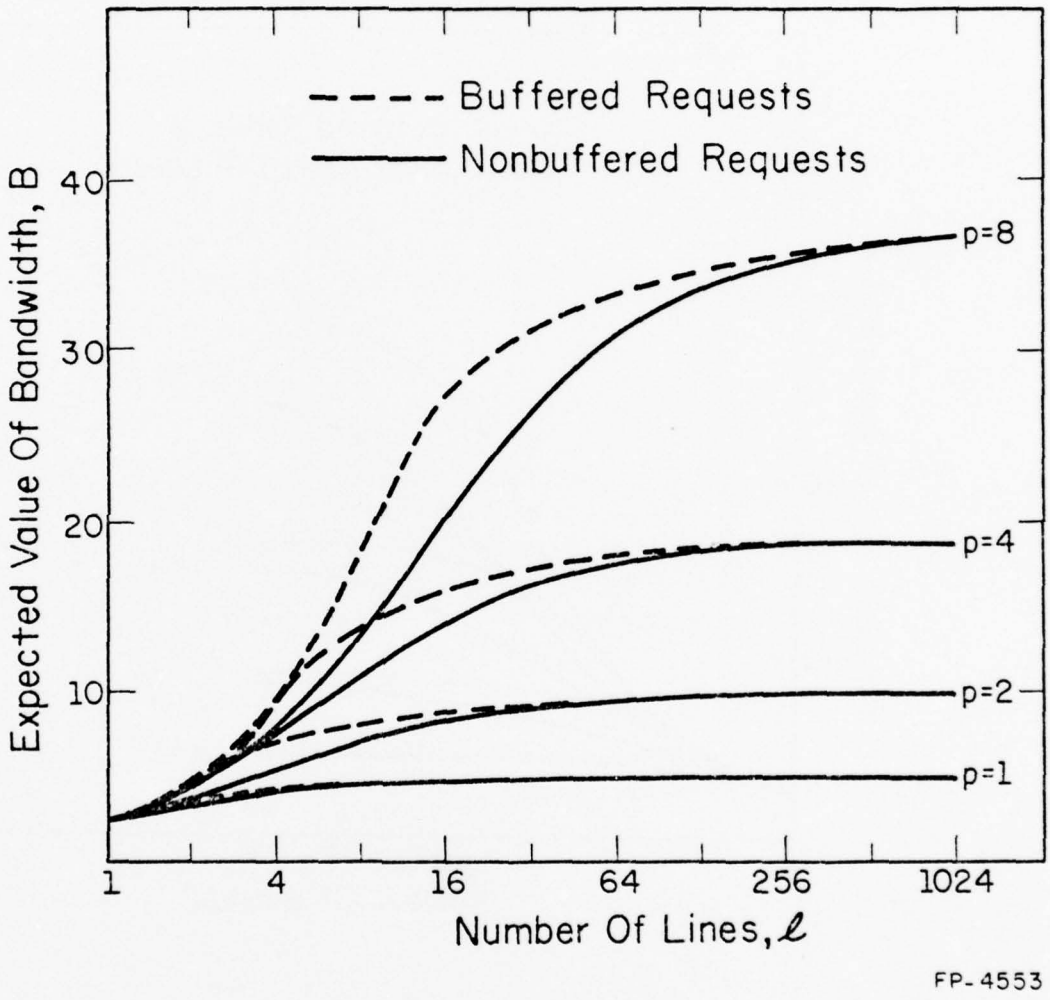


Figure 5.7.4 Bandwidth Versus Memory Configuration for Various Values of p (number of processors), where N = 1024 and $(a_e, c_e) = (2, 4.67)$

figure 5.7.5 for N = 1024. As can be seen from figures 5.7.3 and 5.7.4, buffering tends to have its maximum effect near ℓ = ap, where the upper bound on the bandwidth from theorem 3.6.4 is very weak for the nonbuffered case. For small ℓ and large N, the bandwidth of the nonbuffered case is close to the bound, implying line saturation. In this case, buffering has little effect. For large ℓ and N, P_A approximates 1 in the nonbuffered case as shown in figure 5.7.2. Again, buffering cannot improve P_A significantly. However, if p is large so that cp is close to N, buffering can improve the bandwidth significantly when the lines are not saturated, as shown in figure 5.7.3.

All the parameters discussed in earlier sections affect the performance in the buffered scheme as they do in the nonbuffered scheme. For example, figures 5.7.6 and 5.7.7, when compared with 5.7.3 and 5.7.4, illustrate the effect of reducing the address cycle, a, from 2 to 1. It is seen that this change increases the bandwidth for small values of ℓ in both schemes. However, since c remains constant, the maximum bandwidths are identical for a = 1 and a = 2 and occurs at ℓ = N. In effect, the bandwidth curves for a = 1 level off much earlier than their counterparts for a = 2. The overall effect of reducing a is to achieve a higher bandwidth for configurations with small ℓ.

Buffering can thus most effectively be used for two purposes. One, to increase the performance of a configuration with ℓ in the vicinity of ap. Two, when ℓ exceeds ap, buffering may be used to maintain bandwidth while reducing ℓ. In the nonbuffered case, a rejected request cannot make a retrial in less than an instruction cycle length, whereas, the interval between retrials of a rejected request in the buffered case is

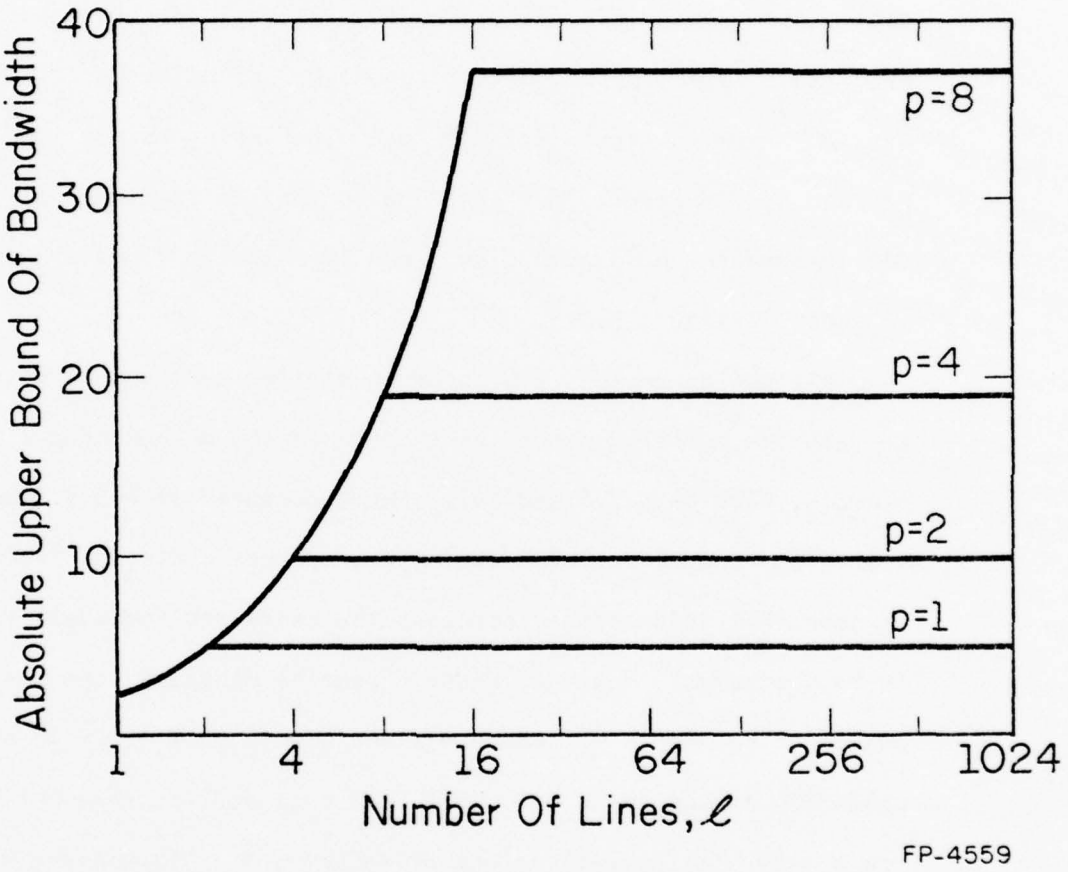


Figure 5.7.5 Maximum Bandwidth Versus Memory Configuration for Various Values of p (number of processors)

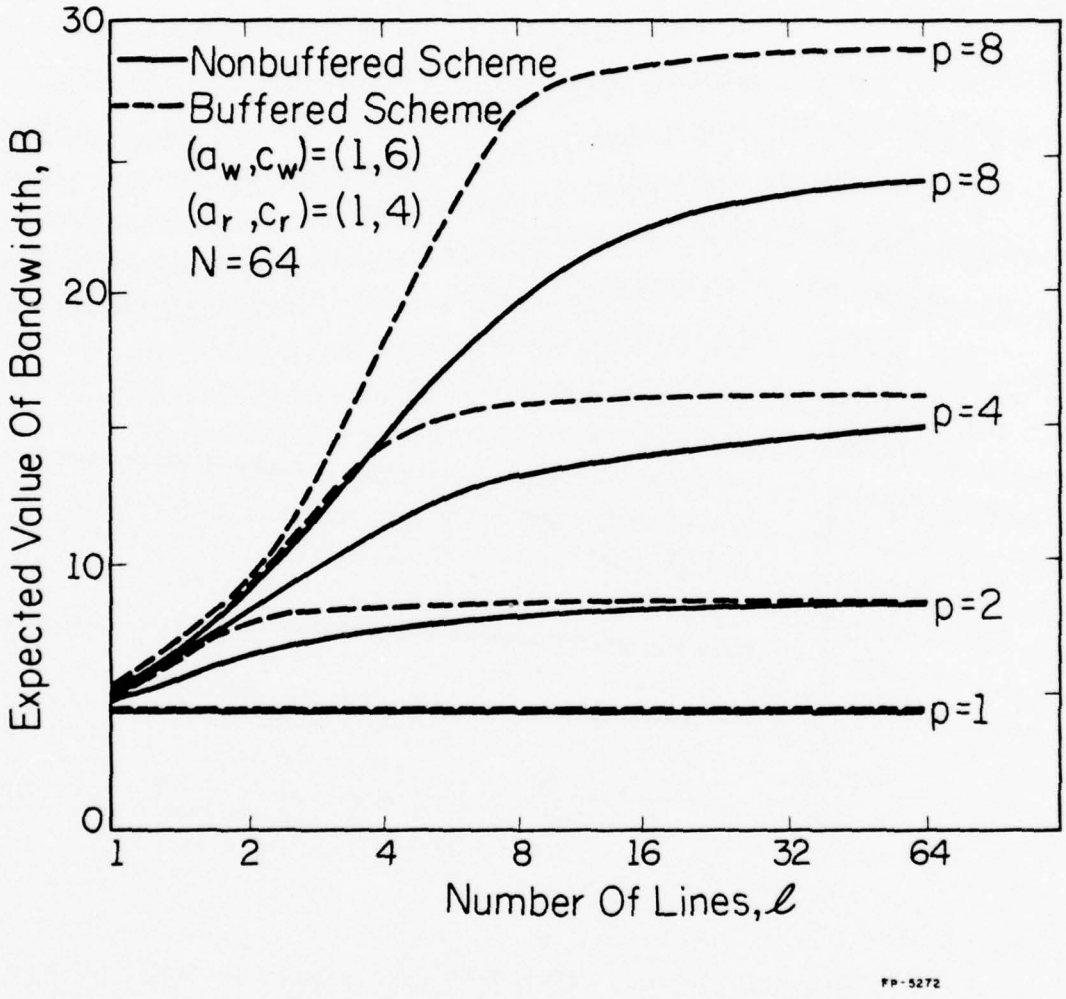


Figure 5.7.6 Bandwidth Versus Memory Configuration for Various Values of p, where N = 64 and $(a_e, c_e) = (1, 4.67)$

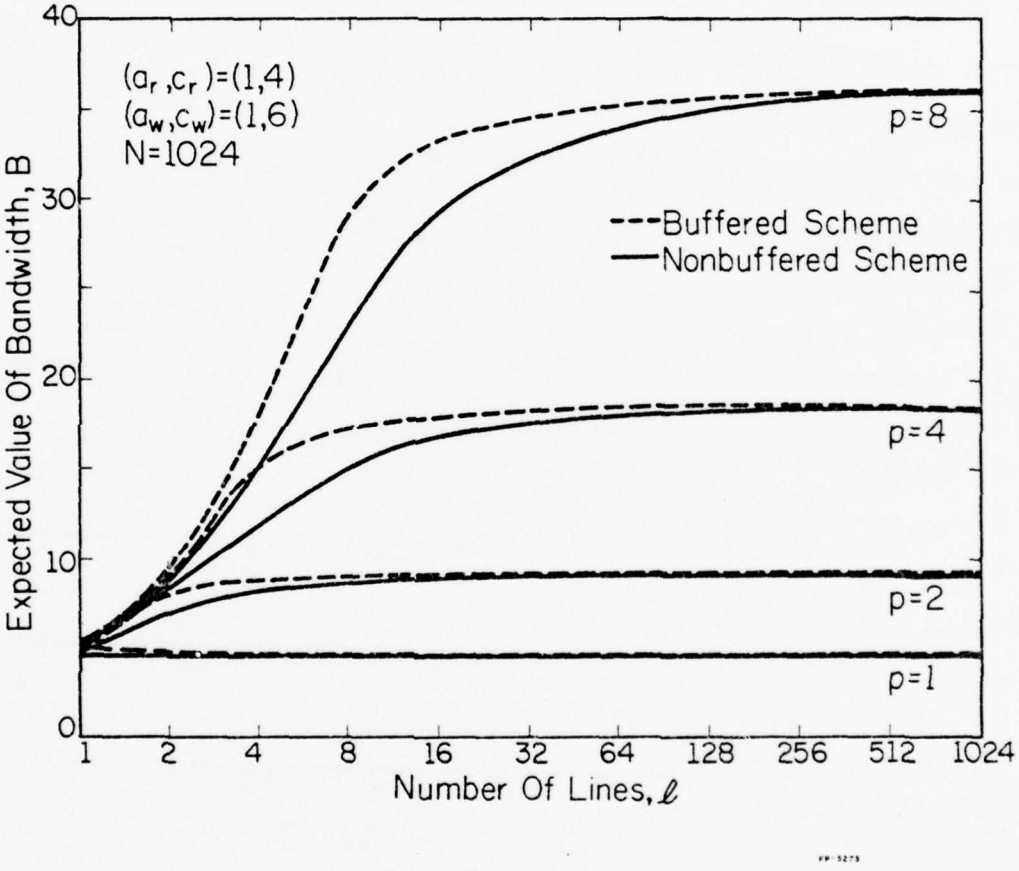


Figure 5.7.7 Bandwidth Versus Memory Configuration for Various Values of p, where N = 1024 and $(a_e, c_e) = (1, 4.67)$

less than an instruction cycle. Hence buffering has a smaller expected average wait time of a request, $\varepsilon(W)$, as evidenced in the examples of figures 5.7.8 and 5.7.9. These curves which are derived from simulations are shown for the more stable region, namely $\ell \geq p$, for which the simulation reaches steady state easily. As N increases, $\varepsilon(W)$ decreases. Buffering also tends to have its maximum effect on $\varepsilon(W)$ for ℓ in the vicinity of ap.

Although the maximum queue length allowable for each processor in the simulation model is infinite, the expected average queue length, $\varepsilon(Q)$, is quite small in these examples. Figures 5.7.10 and 5.7.11 illustrate the expected average queue lengths for various memory configurations and processor orders with $(a_e, c_e) = (2, 4{\cdot}67)$ and N = 64 and N = 1024 respectively. Increasing N decreases $\varepsilon(Q)$ especially for large ℓ. The read and write request distribution also affects the queue length as explained in section 4.3.

5.8 Design Tradeoffs

In this section, certain concepts for designing a parallel-pipelined processor-memory system are introduced by way of examples. Various options open to the designer in meeting a required design objective are investigated. Since the number of parameters is large, the number of available options is so large that it would be impractical to consider all the possible options and derive a rigorous design methodology that can be applicable to the general case. However, a designer can make appropriate decisions at various stages of the design to reduce the number of options and make the design problem less difficult to handle.

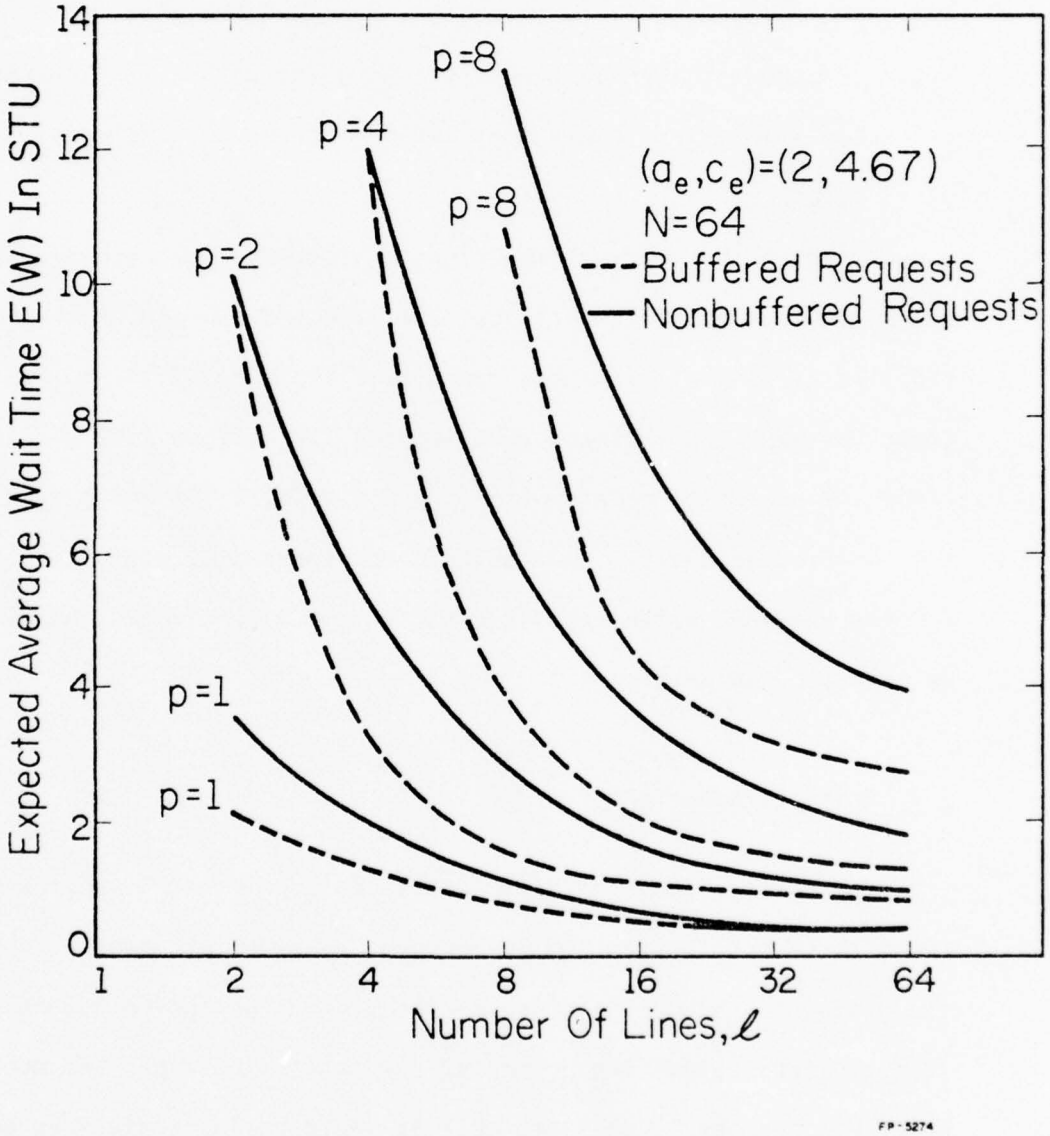


Figure 5.7.8 Wait Time Versus Memory Configuration for Various p, where N = 64 and $(a_e, c_e) = (2, 4.67)$

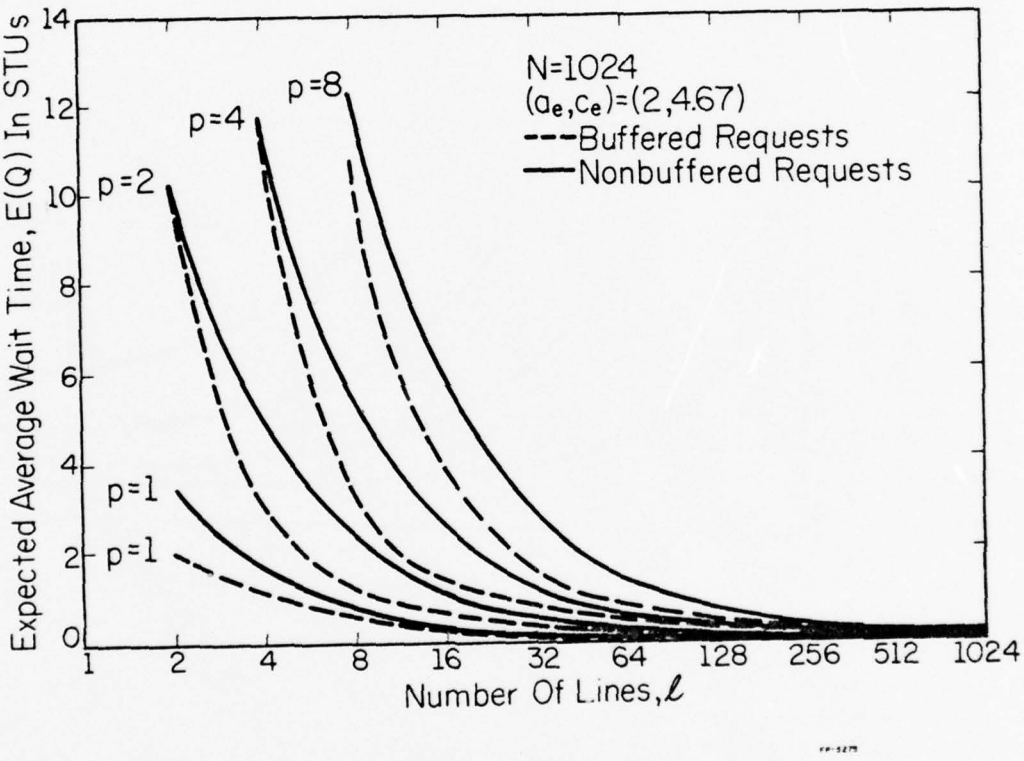


Figure 5.7.9 Wait Time Versus Memory Configuration for Various p, where N = 1024 and $(a_e, c_e) = (2, 4.67)$

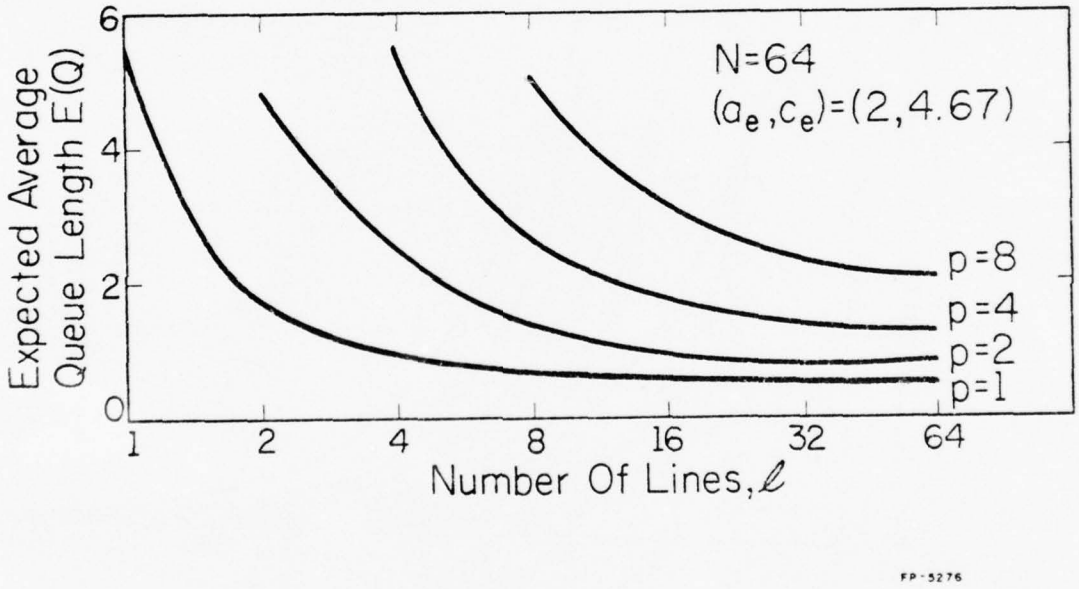


Figure 5.7.10 Queue Length Versus Memory Configuration for Various p, where N = 64 and $(a_e, c_e) = (2, 4.67)$

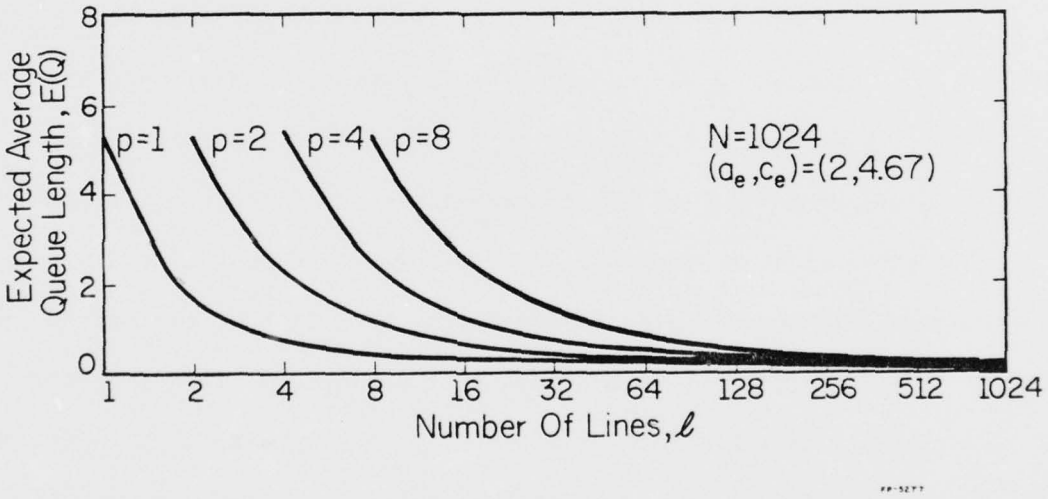


Figure 5.7.11 Queue Length Versus Memory Configuration for Various p, where N = 1024 and $(a_e, c_e) = (2, 4.67)$

Often the cost-performance of a design is the criterion used to evaluate the merit of the design. A design may have very high performance, but may not be economically feasible. Furthermore, it may not be technologically attainable. For example, as the segment time unit, τ, approaches zero, the bandwidth approaches infinity. Eventually prohibitive cost yields to technological impossibility as performance is driven higher. Economic factors and technological constraints must be considered.

An increase in the number of lines, ℓ, requires a more expensive line decoder to decode the line addresses. Recall that the number of gates in a p by ℓ crossbar switch system is proportional to $p\ell$. Hence an increase in ℓ or p increases the cost of the crossbar switch matrix. In addition, an increase in p increases the cost of the processor system, although it also increases the bandwidth of the system. From previous discussion, it was seen that an increase in ℓ increases the probability of acceptance of a request. An increase in ℓ increases the number of line drivers that are required to drive the module addresses down the line. Line drivers are also required to drive the data on the data buses. The line driver capability depends on the number of memory modules, m, on each line. As m increases, the required line driver capability must increase and may result in a higher cost for each line driver.

An increase in the total number of memory modules, N, increases the performance, but may not significantly increase the total cost of the memory of size M (= Nz), because, for semiconductor memory, the cost per bit is virtually constant for a wide range of module sizes, z. However,

the increase in N requires a larger decoder which is used to decode the memory address to select the addressed memory module. Hence an increase in N may increase the cost of the memory address decoder. The choice of a, c, ℓ, N and p also affect the size and hence the cost of the function unit that is required to accept or reject an incoming request which may encounter a line or module collision.

Such contrasting requirements tend to pose a limiting constraint on the design parameters.

In some semiconductor memory modules, the absolute module characteristics (a_o, c_o) are such that $a_o = c_o$. However, with present trends in technology, it has been possible to reduce the address cycle time, a_o such that $a_o < c_o$. It was shown that the probability of acceptance for a given τ, increases with decrease in a_o for small values of ℓ. In some semiconductor LSI memories, for example, in the MOS technology, a_o has been significantly reduced without significant increase in the cost per bit of the memory chip. The memory cycle, c_o, has been decreasing with technological advancement. However, the cost per bit of the memory chip does not necessarily increase with decreases in c_o. Hence it may be assumed that for a certain range of c_o, the cost per bit is virtually constant. However, in general, the cost per bit of a memory chip is related to the cycle time.

Three case studies are discussed to illustrate some design tradeoffs. One, achieves at least a bandwidth, B_o, for the nonbuffered scheme. Two, achieves at least a probability of acceptance, P_A, for good turn-around time. The third case study which has two examples, illustrates the use of buffering. In all cases, the different memory configurations which meet the design constraints are investigated.

In the first example, the main design goal is to achieve at least a bandwidth B_o or better for a parallel-pipelined processor of order (s, p) having access to an (ℓ, m) memory configuration with absolute module characteristics, (a_o, c_o). For this case study, assume that the following parameters are fixed.

1. Processor order (s, p) = (s, 4), s > c.
2. Segment time unit, τ seconds.
3. Module characteristics (a, c) = (1, 4).

Suppose it is required to obtain a bandwidth $B \geq B_o = 15.00$ requests in one memory cycle. Hence, $P_A \geq 15/16 \triangleq 0.94$. Assume also that a nonbuffered scheme is preferred, then theorem 3.5.2 can be applied since a = 1. Below are lists of possible memory configurations and their corresponding performances. This table is constructed by choosing N and finding a minimum ℓ that obtains a $P_A \geq 0.94$.

Total number of modules N	Memory configuration (ℓ, m)	Probability of acceptance $P_A(a, c, p) = P_A(1, 4, 4)$
256	(128, 2)	0.95
512	(64, 8)	0.96
1024	(32, 32)	0.94
∞	(32, ∞)	0.95

Notice that all the above memory configurations produce a probability of acceptance of at least 0.94 as required. Also notice that increasing the total number of memory modules, N, beyond 1024 cannot reduce the number of lines, ℓ, below 32 while maintaining a P_A of at least 0.94. In order to arrive at a cost-effective design, the cost factors

of the various design options trading off N against ℓ have to be taken into consideration. If ℓ is very critical and needs to be reduced below 32, the buffering scheme will have to be used, since a is 1 and cannot be reduced further.

As a second case study, suppose P_A is required to be at least 0.94 in order to obtain a good turn-around time. Assume that the following parameters are given.

1. Processor order (s, p), such that $s \cdot p = 28$.
2. $N = 1024$.
3. $(a, c) = (1, c)$, such that $1 \leq c < s$.
4. The segment time unit, τ, is fixed.

Suppose again that the nonbuffered system is preferred. Below is a list of the possible processor-memory configurations that achieve the required P_A so that the number of distinct instruction streams, $sp = 28$. This table is constructed by choosing all possible s and p combinations such that $s \cdot p = 28$ and $s > c$. Assuming that $c = s - 3$, find the minimum ℓ such that $P_A \geq 0.94$. Then, $m = N/\ell$.

Processor Order (s, p)	Memory cycle c	Memory configuration (ℓ, m)	Bandwidth No. accepted requests/STU
(4, 7)	1	(64, 16)	7 x 0.94
(7, 4)	4	(32, 32)	4 x 0.94
(14, 2)	11	(16, 64)	2 x 0.94
(28, 1)	25	(1, 1024)	1 x 0.94

If the smallest ℓ is desirable, the processor order is (s, p) = (28, 1) and the memory configuration is (ℓ, m) = (1, 1024). For this combination, c = 25. Hence if τ = 50 ns, then the absolute memory cycle

$c_o = c\tau = 1250$ ns. All the above options satisfy the P_A requirement and again the cost-effectiveness of each combination should be considered. It is interesting to see how the bandwidth varies with the processor-memory configuration. As the bandwidth column indicates, the best performance is obtained by the processor order (s, p) = (4, 7) having access to the (64, 16) memory configuration. Notice that in this case study, the bandwidth is proportional to the number of processors, p. Hence there is a freedom of choice of bandwidth since designs with a variety of B are attainable.

The third case study illustrates the use of buffering to reduce the number of lines for various systems. Two examples are given. For the first example, assume that the objective is a minimum number of lines, ℓ, such that the bandwidth, $B \geq 14.00$ requests per memory cycle. Assume also that the following parameters are fixed.

1. Processor order, (s, p) = (5, 4).
2. Two effective module characteristics are obtainable namely $(a_e, c_e) = (1, 4{\cdot}67)$ and $(a_e, c_e) = (2, 4{\cdot}67)$.
3. The segment time unit, τ, is fixed.

Memory configurations with two possible memory sizes are investigated, namely, N = 64 and N = 1024. Figures 5.8.1 and 5.8.2 show, for the BRP and NRP systems, the bandwidth versus memory configurations for $(a_e, c_e) = (1, 4{\cdot}67)$ and $(a_e, c_e) = (2, 4{\cdot}67)$ respectively for N = 64. Similarly, figures 5.8.3 and 5.8.4 illustrate, for the BRP and NRP systems, the bandwidth versus memory configurations for $(a_e, c_e) = (1, 4{\cdot}67)$ and $(a_e, c_e) = (2, 4{\cdot}67)$ respectively for N = 1024. All the above figures were derived from the simulation results. From these curves, the following options which satisfy the bandwidth requirement are obtained as marked on the curves.

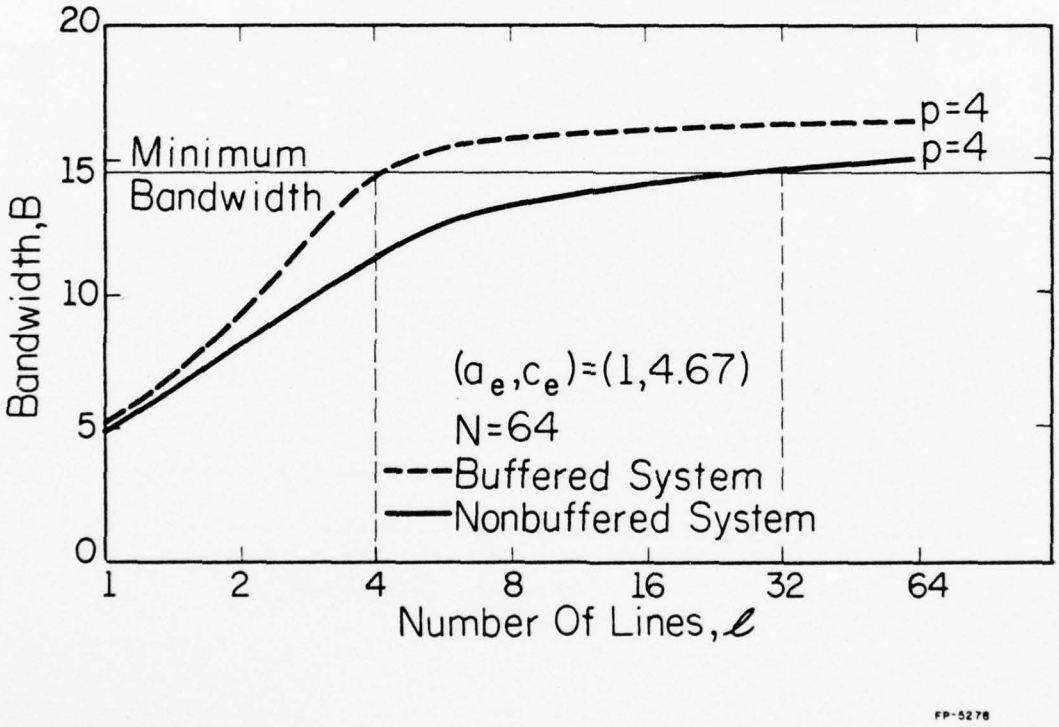


Figure 5.8.1 Bandwidth Versus Memory Configuration for N = 64 and $(a_e, c_e) = (1, 4.67)$

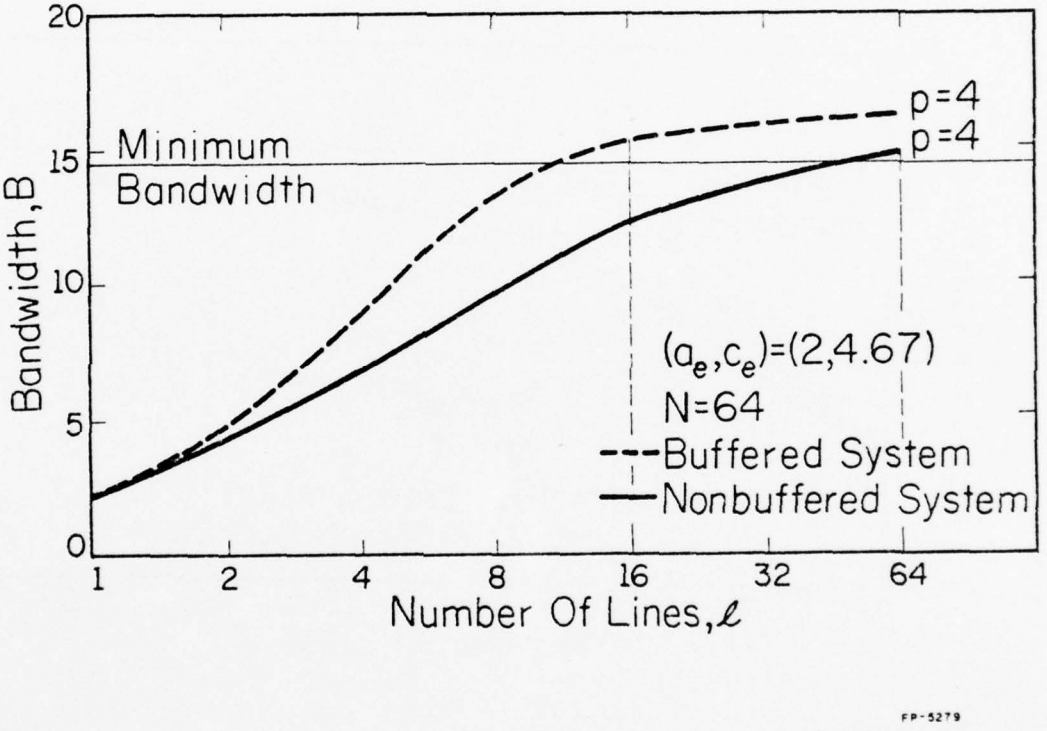


Figure 5.8.2 Bandwidth Versus Memory Configuration for N = 64 and $(a_e, c_e) = (2, 4.67)$

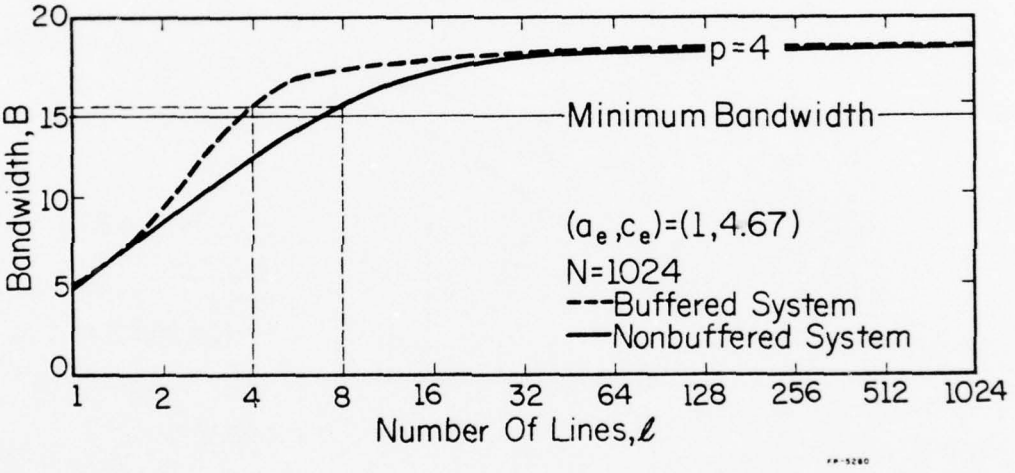


Figure 5.8.3 Bandwidth Versus Memory Configuration for N = 1024 and $(a_e, c_e) = (1, 4.67)$

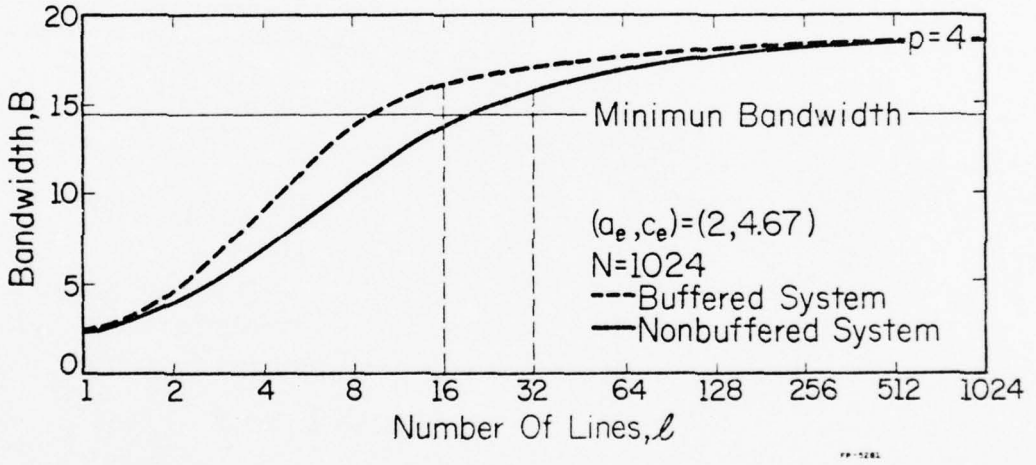


Figure 5.8.4 Bandwidth Versus Memory Configuration for N = 1024 and $(a_e, c_e) = (2, 4.67)$

	a = 1		a = 2	
	NRP	BRP	NRP	BRP
N = 64	ℓ = 32	ℓ = 4	ℓ = 64	ℓ = 16
N = 1024	ℓ = 8	ℓ = 4	ℓ = 32	ℓ = 16

In this example, the memory configuration which gives the **least** ℓ is the BRP system with $(\ell, m) = (4, 16)$ for $(a_e, c_e) = (1, 4{\cdot}67)$. The memory configuration $(\ell, m) = (4, 256)$ also gives the least ℓ. However, in this case, N = 1024 does not reduce ℓ and is thus less cost-effective than the (4, 16) configuration. If modules with module characteristics $(a_e, c_e) = (2, 4{\cdot}67)$ are cheaper than $(a_e, c_e) = (1, 4{\cdot}67)$, then one might consider trading the memory configuration $(\ell, m) = (4, 16)$, using (1, 4·67) module characteristics, for $(\ell, m) = (16, 4)$, using (2, 4·67) module characteristics. In the absence of current technological costs, the solution is not obvious. However, the model provides many choices.

In the second example of this case study, let the bandwidth be at least 18.00 or more. Assume that the parameters are fixed as in the previous example. The memory size with N = 64 is easily ruled out as a possibility, since there exists no configuration with N = 64 that achieves a bandwidth of at least 18.0. The options for N = 1024 are shown below. From figures 5.8.3 and 5.8.4, various options can be obtained as listed below.

	a = 1		a = 2	
	NRP	BRP	NRP	BRP
N = 1024	ℓ = 64	ℓ = 32	ℓ = 256	ℓ = 128

In this example, the memory configuration with the least number of lines is the BRP system with $(\ell, m) = (32, 32)$ and (a, c) = (1, 4·67). Notice

that for (a, c) = (2, 4·67) buffering also reduces the number of lines.

Hence buffering can indeed be used to reduce the number of lines in some cases while maintaining the bandwidth. However, in addition to buffering costs, the buffering system requires time for update and maintenance of the queues which is a source of overhead incurred by the system. The buffering system, with its benefits of reducing ℓ while keeping performance constant or increasing the performance for ℓ in the vicinity of ap, should be weighed against the overhead incurred in the BRP system in order to arrive at a cost-effective design.

5.9 Burst Mode Operation

So far, we have investigated the effects of various parameters on the performance for parallel-pipelined processor of order (s, p), where p simultaneous requests are issued to the memory system every segment time unit. We have assumed that these p parallel requests are dispatched simultaneously to the arbitration logic to determine which of them can be accepted for service in the memory system. The mode of operation in which p parallel requests are dispatched simultaneously every STU to the accept/reject logic for processing is called multiplex mode.

We have shown that buffering each of the p parallel requests in its associated processor's buffer may improve the performance. In this scheme, the multiplex mode of operation is also in effect, since we have assumed that at most one memory request from each buffer is obtained for service in the memory system every STU. Hence at most p requests are

accepted simultaneously. In this section, we will describe a different buffering scheme which necessitates a new mode of operation. We will also investigate the effect of such schemes on the performance.

In one memory cycle, a total of cp memory requests are issued. These cp memory requests are buffered in a buffer of length cp whereupon the cp memory requests are dispatched at the end of the memory cycle to the accept/reject logic in order to determine which of the cp requests can be accepted for service in the memory. The mode of operation in which the total number of memory requests issued in one memory cycle are dispatched simultaneously to the accept/reject logic at the end of the memory cycle is called burst mode. The multiplex and burst mode operations are illustrated in figure 5.9.1.

If the segment time unit is τ in the multiplex mode, then the effective segment time unit in the burst mode is $\tau' = c\tau$. Notice that the memory configuration and the absolute module characteristics are unaltered by the choice of the mode of operation. Hence every τ' seconds, cp memory requests are dispatched. We will investigate the performance of the burst mode operation and compare it with the multiplex mode operation. It will be shown that for some module characteristics, (a, c), and memory configurations, the burst mode outperforms the multiplex mode and vice versa.

Two extreme cases of module characteristics will be investigated to draw a comparison between the two modes of operation. The probability of acceptance will serve to indicate the performance of the various module characteristics. For a = c, the multiplex mode gives the probability of acceptance, $P_A(a, c, p)$ from theorem 3.5.3 as

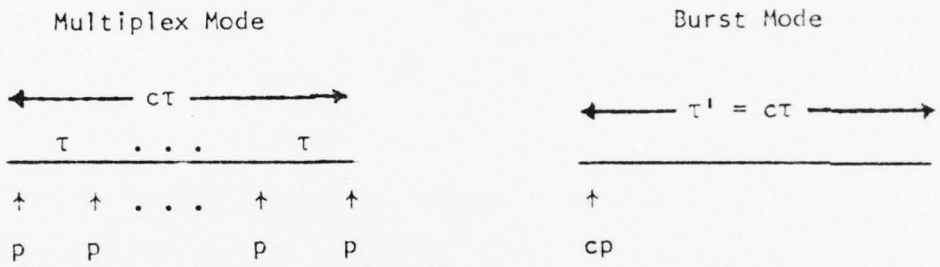
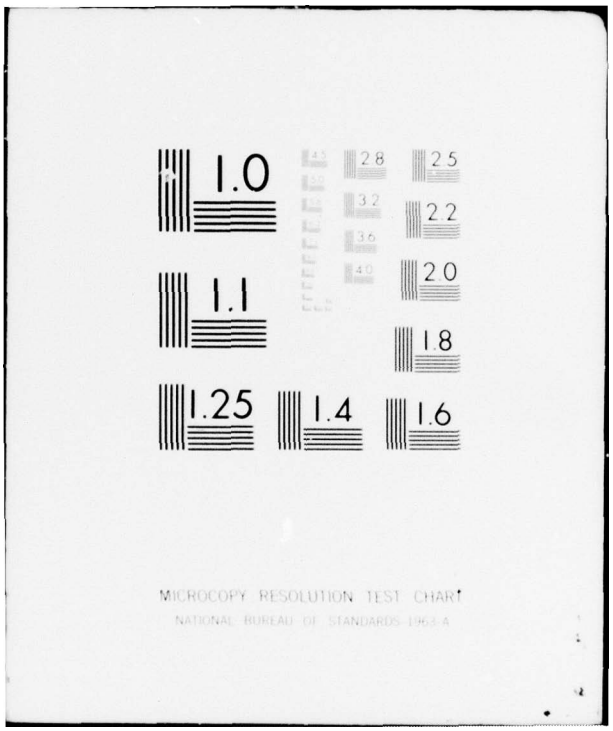


Figure 5.9.1. Multiplex and burst mode sequencing.

AD-A042 646 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
MEMORY ORGANIZATIONS AND THEIR EFFECTIVENESS FOR MULTIPROCESSIN--ETC(U)
MAY 77 F A BRIGGS DAAB07-72-C-0259
UNCLASSIFIED R-768 NL

3 OF 3
AD A042646

END
DATE FILMED
8-77
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

$$P_A(c, c, p) = \frac{[1 - (1 - \frac{1}{\ell})^p]\ell/p}{1 + [1 - (1 - \frac{1}{\ell})^p](c-1)}$$

Since the segment time unit for the burst mode is τ', which is equal to the absolute memory cycle of the module, $c\tau$, the relative address and memory cycles are both equal to 1. Since the number of parallel requests dispatched every τ' seconds is cp, the probability of acceptance for the burst mode is, from theorem 3.5.2,

$$P_A(1, 1, cp) = [1 - (1 - \frac{1}{\ell})^{cp}] \frac{\ell}{cp} .$$

Theorem 5.9.1 For $a = c$, $P_A(1, 1, cp) \geq P_A(c, c, p)$, for all memory configurations.

Proof: We must show that

$$[1 - (1 - \frac{1}{\ell})^{cp}] \frac{\ell}{cp} \geq \frac{[1 - (1 - \frac{1}{\ell})^p] \frac{\ell}{p}}{1 + [1 - (1 - \frac{1}{\ell})^p](c - 1)} .$$

For $\ell = 1$, it is trivial to show that $P_A(1, 1, cp) = P_A(c, c, p)$. Assume that $\ell > 1$. Let $y = (1 - \frac{1}{\ell})^p$. Since $p \geq 1$, for $\ell > 1$, $y < 1$. Then we must show that

$$(1 - y^c) \geq \frac{(1 - y)c}{1 + (1 - y)(c - 1)} = \frac{1 - y}{1 - y + \frac{y}{c}} .$$

That is, we need to show that

$$(1 - y^c)(1 - y + \frac{y}{c}) \geq (1 - y).$$

Multiplying the left hand side out, we obtain

$$1 - y^c - y + y^{c+1} + \frac{y}{c} - \frac{y^{c+1}}{c} = 1 - y + y^c(y - 1) + \frac{y}{c}(1 - y^c).$$

Hence we need to show that

$$y^c(y - 1) + \frac{y}{c}(1 - y^c) \geq 0, \quad \text{or,} \quad \frac{y}{c}(1 - y^c) \geq (1 - y)y^c,$$

or, $\frac{(1 - y^c)}{y^{c-1}(1 - y)} \geq c.$

However, $\frac{1 - y^c}{1 - y} = 1 + y + y^2 + \ldots + y^{c-2} + y^{c-1}.$

Hence $\frac{1 - y^c}{y^{c-1}(1 - y)} = \frac{1}{y^{c-1}} + \frac{1}{y^{c-2}} + \frac{1}{y^{c-3}} + \ldots + \frac{1}{y} + 1 = \sum_{i=0}^{c-1} \frac{1}{y^i}$

Since $y < 1$, $\frac{1}{y^i} \geq 1$, for $0 \leq i \leq c - 1$.

Hence, $\sum_{i=0}^{c-1} \frac{1}{y^i} > c$, and the theorem holds. □

Therefore, for module characteristics (a, c), such that a = c, the burst mode of operation has as good or better probability of acceptance then the multiplex mode of operation for any memory configuration. Notice that the bandwidths expressed as the number of memory requests accepted per memory cycle, for the burst and multiplex modes are $cpP_A(1, 1, cp)$ and $cpP_A(c, c, p)$ respectively. Hence the bandwidth of the burst mode is also as good or better than that of the multiplex mode. In addition to the superior performance of the burst mode, it eliminates the need for the functional unit which is otherwise required to test for busy line collisions. However, the cp by ℓ crossbar switch required for the burst mode is more complex than the p by ℓ crossbar switch for the multiplex mode. This complexity may significantly increase the cost of implementing the burst mode for c > 1. Notice that for c = 1, the burst and multiplex modes are equivalent. The burst mode may also require a large number of tests to detect multiple access line collisions, since cp may be large. Furthermore, the bus widths and buffering costs are higher for the burst mode.

Notice however, that for the burst mode, the expected number of busy lines at the end of a memory cycle is zero, since c = 1 for the burst mode. Hence all the ℓ lines are always available for service when the cp parallel requests are dispatched simultaneously.

The other extreme case to be considered is the memory system with module characteristics (a, c) = (1, c).

Theorem 5.9.2 For a = 1, $P_A(1, 1, cp) \geq P_A(1, c, p)$, if $\ell = N$. □

The proof follows immediately by substituting ℓ for N in the expression for

$$P_A(1, c, p) = \frac{[1 - (1 - \frac{1}{\ell})^p] \frac{\ell}{p}}{1 + [1 - (1 - \frac{1}{\ell})^p] \frac{\ell}{p} \cdot \frac{p(c-1)}{N}}$$

and applying theorem 5.9.1.

Thus for the case a = 1, $\ell = N$, the burst mode of operation is as good or better than the multiplex mode. For $\ell < N$, a proof has not been obtained. However, the following hypothesis seems to hold.

Hypothesis 5.9.1 For a = 1, $P_A(1, 1\ cp) \leq P_A(1, c, p)$, if $\ell < N$. □

This hypothesis has been shown to be true for a wide variety of memory configurations (ℓ, m), and module characteristics (1, c). Assuming that this hypothesis is true in general, the multiplex mode exhibits as good or better performance than the burst mode when a = 1 and $\ell < N$.

The above theorems and hypothesis, suggest that for module characteristics (a, c), such that 1 < a < c, there may exist some memory

configurations for which $P_A(a, c, p) > P_A(1, 1, cp)$ and some for which $P_A(a, c, p) < P_A(1, 1, cp)$. Examples of the former may be for small a and $\ell < N$. Examples of the latter may be for large a or $\ell = N$.

Although in some cases the burst mode may exhibit a better performance than the multiplex mode, its cost-effectiveness is questionable. The burst mode operation requires that all incoming memory requests within a memory cycle be buffered and dispatched simultaneously at the end of the memory cycle to the memory system. This mode of operation may affect the synchronous nature of operation in the pipelined processor and operands may have to be buffered and a scheduling scheme introduced to maintain correct sequencing of the processes.

6. CONCLUSIONS

6.1 Summary of Results

The object of this research is to develop a flexible semiconductor memory organization for parallel-pipelined processors and investigate the effect of memory interference on the system performance for a variety of module characteristics, processor orders and memory configurations. We exploited the characteristics of semiconductor memories to develop a general yet flexible memory organization for multiprocessor systems. We have fully investigated the effect of the memory interference on the system performance for two classes of module characteristics, namely, (a, c) such that $a = 1$ and $c \geq 1$, and (a, c) such that $a > 1$ and $a \leq c \leq 2a$. We discovered that the complexity of the Markov analysis grows nonlinearly with $\left\lfloor \frac{c-1}{a} \right\rfloor$ for $a > 1$. Hence we do not have a general solution of performance for arbitrary (a, c).

In chapter two, the memory organization was described. It was seen that the memory interference problem is more complicated than models developed by previous investigators. However, the model lends itself to Markov analysis.

In chapter three, we developed the analytical model and introduced the probability of acceptance of a request to evaluate the effect of memory interference on the system performance. Since a general expression for $P_A(a, c, p)$ is not known for any module characteristics, (a, c), we have obtained the lower and upper bounds on $P_A(a, c, p)$. We observed that the performance of the analytic model is very weak for memory

configurations (ℓ, m), such that ℓ is in the vicinity of ap. Furthermore, it was found that for any memory configuration, the module characteristics with $a = 1$ give the best performance.

In chapter four, we investigated simulation models for the buffered and nonbuffered request processor system. In the nonbuffered request processor system, rejected requests were resubmitted one instruction cycle later with the same address. We demonstrated that the results of the experimental model were not significantly different from the results of the analytical model. This justifies the assumption that the discarding of rejected requests, for analytical purposes, does not necessitate a significant deviation of the analytical model from reality. The buffered scheme was shown to be as good or better than the nonbuffered scheme for any memory configuration and module characteristics.

In chapter five, the effects of the various parameters on performance were investigated. We found that for very large number of memory modules, N, the effect of the memory cycle, c, is insignificant.

There is generally less payoff to increasing N for large ℓ and overall p, and for small ℓ and large p. However, there is significant payoff to increasing N for small ℓ and small p, and for large ℓ and large p.

Memory configurations where $\ell < p$ give poor performance. The performance deteriorates as the address cycle, a, is increased in this region. For $\ell = p$, there is a point of inflection. For $a > 1$, there is a significant payoff to increasing ℓ when ℓ lies between p and ap.

The effect of module characteristics on performance is minimal when ℓ and N are sufficiently large. We have shown that for small ℓ, the effect of the address cycle, a, can be drastic. In general, a is usually the

critical factor when ℓ is small and N is large. Hence when possible, a module characteristic with a = 1 should be chosen. Furthermore, we showed that for ℓ = N, the performance is independent of the address cycle, a.

The processor speed is another critical factor that determines the bandwidth of the system. We have shown some illustrations of the effect of the processor speed on the performance.

Buffering has its maximum effect on performance when the memory configuration is such that ℓ = ap. However, for ℓ < p, and for large ℓ and N, buffering tends to have very little effect on performance. Hence buffering can be effectively used for two purposes. One, to increase the performance for ℓ in the vicinity of ap. Two, if ℓ > ap, buffering can be used to reduce the number of lines, ℓ, while maintaining the bandwidth.

We have shown, by some design tradeoff examples, that there exists a wide variety of design options open to the designer. However, the designer should make judicious choices at each design step to optimize the cost-effectiveness of the design.

6.2 Suggestions for Further Research

No attempt has been made to develop an analytical model for the buffered request processor system. Although we have shown that the complexity of the analysis for the nonbuffered scheme can become too difficult, it may be possible to develop some approximate queueing models for the buffered scheme.

A possible extension of this thesis may be directed dowards developing a model for dynamic memories and investigating their performance in multiprocessor systems. It may be possible to characterize the dynamic memory module as a function of the access time distributions which depend on the module size, cell interconnection patterns and allowable memory transformations.

Software development for multiprocessor systems that utilize the memory organization discussed in this thesis should be investigated. Memory allocation for different processes may also affect the performance significantly. This problem may be interesting. In practice job sequencing in such computer systems may be complex.

Possible pipelining of the interconnection network between processor and memory could significantly increase the efficiency of detecting memory collisions and routing accepted requests to their respective addressed modules and should be studied.

Finally the interrelationship of the results presented here and the design and management of a memory hierarchy is a most important problem.

APPENDIX A

In order to calculate the probability of acceptance, $P_A(a, c, p)$ for the system with module characteristics $(a, c) = (2, 4)$ and processor order, $p = 1$, from the system state graph, we will reproduce the reduced system state graph $G_s'(a, c)$, for module characteristics $(a, c) = (2, 4)$. $G_s'(2, 4)$ is shown in figure A.

Let P_A, P_B, P_C, P_D, P_E, P_F, P_G and P_I denote the steady state probability of being in system states [Ø], [(1)], [(2)], [(1)(2)], [(3)], [(1)(3)], [(2)(3)] and [(1)(2)(3)] respectively. We can therefore proceed to write the steady state equations for $G_s'(2, 4)$ as follows, noting that $N = \ell m$.

$$P_A = \frac{1}{N} P_E \quad \ldots \quad 1.$$

$$P_B = P_A + \frac{N-1}{N} P_E + \frac{m-1}{N} (P_C + P_G) \quad \ldots \quad 2.$$

$$P_C = \frac{1}{\ell} P_B + \frac{m+1}{N} P_F \quad \ldots \quad 3.$$

$$P_D = \frac{\ell-1}{\ell} P_B + \frac{m-1}{N} (P_D + P_I) + \frac{N-m-1}{N} P_F \quad \ldots \quad 4.$$

$$P_E = \frac{1}{N} P_C + \frac{2}{N} P_G \quad \ldots \quad 5.$$

$$P_F = \frac{\ell-1}{\ell} P_C + \frac{N-m-1}{N} P_G \quad \ldots \quad 6.$$

$$P_G = \frac{m+1}{N} P_D + \frac{m+2}{N} P_I \quad \ldots \quad 7.$$

$$P_I = \frac{\ell-2}{\ell} P_D + \frac{N-2m-1}{N} P_I \quad \ldots \quad 8.$$

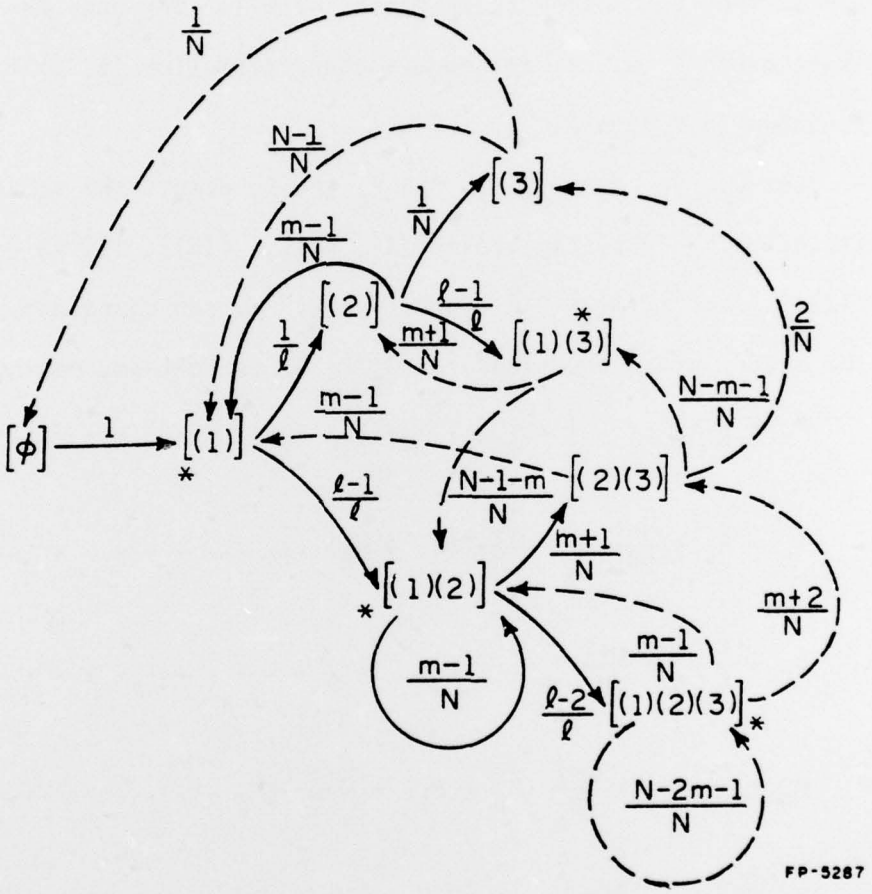


Figure A. Reduced System State Graph, $G'_s(2, 4)$

$P_A(2, 4, 1)$ is the probability of being in any of the system acceptance states in $G_s'(2, 4)$. The system acceptance states in $G_s'(2, 4)$ are [(1)], [(1)(2)], [(1)(3)] and [(1)(2)(3)]. Hence,

$$P_A(2, 4, 1) = P_B + P_D + P_F + P_I \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad 9.$$

Moreover, since a module would have to be in one of the systems states,

$$P_A + P_B + P_C + P_D + P_E + P_F + P_G + P_I = 1 \quad \ldots\ldots\ldots\ldots \quad 10.$$

Substituting for P_A in equation 2,

$$P_B = P_E + \frac{m-1}{N} (P_C + P_G) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad 11.$$

Substituting for P_E from equation 5 in P_B expression,

$$P_B = \frac{1}{N} P_C + \frac{2}{N} P_G + \frac{m-1}{N} (P_C + P_G)$$

Simplifying,

$$P_B = \frac{m}{N} P_C + \frac{m+1}{N} P_G \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad 12.$$

Substituting for P_B and P_F in equation 3,

$$P_C = \frac{m}{N} \left(\frac{m}{N} P_C + \frac{m+1}{N} P_G\right) + \frac{m+1}{N} \left(\frac{N-m}{N} P_C + \frac{N-m-1}{N} P_G\right)$$

Simplifying,

$$P_C = \frac{m+1}{N-m} P_G \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad 13.$$

Substituting the P_C expression from equation 13 in 12,

$$P_B = \left(\frac{m}{N} \cdot \frac{m+1}{N-m} + \frac{m+1}{N} \cdot \frac{N-m}{N-m}\right) P_G$$

Therefore,

$$P_B = \frac{m+1}{N-m} P_G \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad 14.$$

Notice that, $P_B = P_C$.

Substituting the P_C expression from equation 13 in 5,

$$P_E = \left(\frac{1}{N} \cdot \frac{m+1}{N-m} + \frac{2}{N} \cdot \frac{N-m}{N-m}\right) P_G .$$

Therefore,

$$P_E = \frac{2N-m+1}{N(N-m)} P_G \quad \ldots \quad 15.$$

Substituting the P_E expression from 15 in 1,

$$P_A = \frac{2N-m+1}{N^2(N-m)} P_G \quad \ldots \quad 16.$$

Substituting the P_C expression from 13 in 6,

$$P_F = \left(\frac{N-m}{N} \cdot \frac{m+1}{N-m} + \frac{N-m-1}{N} \cdot \frac{N-m}{N-m}\right) P_G .$$

Simplifying,

$$P_F = P_G \quad \ldots \quad 17.$$

From equation 8,

$$P_I = \frac{N-2m}{2m+1} P_D \quad \ldots \quad 18.$$

Substituting P_B, P_F and P_I expressions from 14, 17 and 18 into 4,

$$P_D = \frac{N-m}{N} \cdot \frac{m+1}{N-m} P_G + \frac{m-1}{N} \left(1 + \frac{N-2m}{2m+1}\right) P_D + \frac{N-m-1}{N} P_G$$

Simplifying,

$$P_D = P_G + \frac{(N+1)(m-1)}{N(2m+1)} P_D$$

Simplifying further,

$$P_D = \frac{N(2m+1)}{Nm+2N-m+1} P_G \quad \ldots \quad 19.$$

Hence from equation 18,

$$P_I = \frac{N(N-2m)}{Nm+2N-m+1} P_G \quad \ldots \quad 20.$$

Substituting P_A, P_B, P_C, P_D, P_E, P_F and P_I expressions from 16, 14, 13, 19, 15, 17, and 20 respectively in 10,

$$\frac{2N-m+1}{N^2(N-m)} + \frac{2(m+1)}{N-m} + \frac{N(2m+1)}{Nm+2N-m+1} + \frac{2N-m+1}{N(N-m)} + 2 + \frac{N(N-2m)}{Nm+2N-m+1} = \frac{1}{P_G}$$

Expanding and simplifying,

$$P_G = \frac{N^2(N-m)(Nm-2N-m+1)}{(N^2+Nm+2N-m+1)(N^2+2N-m+1)(N+1)} \quad \ldots\ldots\ldots\ldots\ldots \quad 21.$$

Substituting P_B, P_D, P_F and P_I expressions from 14, 19, 17, and 20 respectively in 9,

$$P_A(2, 4, 1) = \frac{m+1}{N-m} P_G + \frac{N(2m+1)}{Nm+2N-m+1} P_G + P_G + \frac{N(N-2m)}{Nm+2N-m+1} P_G$$

Simplifying,

$$P_A(2, 4, 1) = \frac{(N^2+2N-m+1)(N+1)}{(N-m)(Nm-2N-m+1)} P_G \quad \ldots\ldots\ldots\ldots\ldots \quad 22.$$

Substituting P_G expression from 21 in 22,

$$P_A(2, 4, 1) = \frac{(N^2+2N-m+1)(N+1)}{(N-m)(Nm-2N-m+1)} \cdot \frac{N^2(N-m)(Nm-2N-m+1)}{(N^2+Nm+2N-m+1)(N^2+2N-m+1)(N+1)} .$$

Cancelling common terms in numerator and denominator,

$$P_A(2, 4, 1) = \frac{N^2}{N^2+Nm+2N-m+1} \quad \ldots\ldots\ldots\ldots\ldots \quad 23.$$

It is obvious that deriving an expression for $P_A(2, 4, 1)$ in terms of N and m only is very tedious. This technique cannot be applied easily to a system state graph with a large number of states. Some computer assistance will probably be required as the sparse matrix gets larger for $a > 1$ and $\left\lfloor \frac{c-1}{a} \right\rfloor > 2$.

LIST OF REFERENCES

1. Flynn, M.J., "Very High Speed Computing Systems," Proc. IEEE, Vol. 54, No. 12, pp. 1901-1909, December 1966.

2. Barnes, G.H., "The ILLIAC IV Computer," IEEE Trans. Comput., pp. 746-757, August 1968.

3. Anderson, D.W., et. al., "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," IBM J. of Res. and Dev., pp. 8-24, January 1967.

4. Hintz, R.G., and Tate, D.P., "Control Data STAR-100 Processor Design," Proc. Compcon Fall 72, pp. 1-4, September 1972.

5. Watson, W.J., "The TI ASC - A Highly Modular and Flexible Computer Architecture," Proc. FJCC 1972, pp. 221-228.

6. Hodges, D.A., Semiconductor Memories, IEEE Press, 1972.

7. Davidson, E.S., "Scheduling for Pipelined Processors," Proc. 7th Annual Hawaii Intl. Conf. on Systems Sciences, pp. 58-60, January 1974.

8. Shar, L.E., and Davidson, E.S., "A Multimini-Processor System Implemented Through Pipelining," Computer, Vol. 7, No. 2, pp. 42-51, February 1974.

9. Davidson, E.S., et. al., "Effective Control for Pipelined Computers," Proc. Compcon Spring 75, pp. 181-184, February 1975.

10. Weller, D.L., and Davidson, E.S., "Optimal Searching Algorithms for Parallel-Pipelined Computers," Springer-Verlag Lecture Notes, No. 24, pp. 90-98, August 1975.

11. Danielsson, P-E, and Gudmundsson, B., "Time-Shared Memory-Processor Interface," 1975 Sagamore Comput. Conf. Parallel Processing, pp. 90-98, August 1975.

12. Hellerman, H., Digital Computer System Principles, New York: McGraw-Hill, 1967, pp. 228-229.

13. Knuth, D.E., and Rao, G.S., "Activity in Interleaved Memory," IEEE Trans. Comput., Vol. C-24, No. 9, pp. 943-944, September 1975.

14. Burnett, G.J., and Coffman, E.G., "A Study of Interleaved Memory Systems," 1970 Spring Joint Comput. Conf., AFIPS Conf. Proc., Vol. 36, Montvale, N.J.: AFIPS Press, pp. 467-474, 1970.

15. Burnett, G.J. and Coffman, E.G., "Analysis of Interleaved Memory Systems Using Blockage Buffers," Commun. ACM, Vol. 18, No. 2, pp. 91-95, February 1975.

16. Skinner, C., and Asher, J., "Effect of Storage Contention on System Performance," IBM Syst. J., Vol. 8, No. 4, pp. 319-333, 1969.

17. Strecker, W.D., "Analysis of the Instruction Execution Rate in Certain Computer Structures," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1970.

18. Ravi, C.V., "On the Bandwidth and Interference in Multiprocessors," IEEE Trans. Comput., Vol. C-21, pp. 899-901, August 1972.

19. Bhandarkar, D.P., "Analysis of Memory Interference in Multiprocessors," IEEE Trans. Comput., Vol. C-24, pp. 897-908, September 1975.

20. Sastry, K.V., and Kain, R.Y., "On the Performance of Certain Multiprocessor Computer Organizations," IEEE Trans. Comput., Vol. C-24, pp. 1066-1074, November 1975.

21. Baskett, F., and Smith, A., "Interference in Multiprocessor Computer Systems with Interleaved Memory," Commun. ACM, Vol. 19, No. 6, pp. 327-334, June 1976.

22. Denning, P.J., "The Working Set Model for Program Behaviour," Commun. Ass. Comput. Mach., Vol. 11, pp. 323-333, May 1968.

23. Kleinrock, L., Queueing Systems, Vol. 1: Theory, Wiley - Interscience, 1975.

24. Briggs, F.A., and Davidson, E.S., "Organization of Semiconductor Memories for Parallel-Pipelined Processors," IEEE Trans. Comput., Vol. C-26, pp. 162-169, February 1977.

25. Chang, D., et. al., "On the Effective Bandwidth of Parallel Memories," Dept. of Computer Science, Univ. of Illinois, Urbana, Ill., September 1975.